A Functional Approach to Syntax and Typing

Een wetenschappelijke proeve op het gebied van
de Wiskunde en Informatica

Proefschrift
ter verkrijging van de graad van doctor aan
de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
vrijdag 17 oktober 1997,
des namiddags om 3.30 uur precies

door

Frank Antonius Mathieu van den Beuken

geboren op 10 mei 1970 te Wanssum

druk: Ponsen & Looijen

Promotor: Prof. dr. ir. R.T. Boute
Co-promotor: Dr. ir. H.A. van Thienen

Manuscriptcommissie:
Prof. dr. C. Delgado Kloos (Technical University of Madrid)
Prof. dr. A. Hoogewijs (University of Gent)
Dr. M. Seutter

ISBN 90-9010854-8
NUGI 851

Acknowledgements

I would like to thank my promotor, Prof. dr. ir. Raymond Boute. The Funmath language, which is the main subject of this thesis and in which all formal text in this thesis is written, was conceived by him. I am also grateful to him for the things I learned about the theory and practice of system design and its repercussions on the design of system specification languages. His detailed comments on draft versions of this thesis have resulted in many improvements.

I would like to thank my co-promotor, Huub van Thienen, for his intensive thesis supervision during the final years of this research. His comments on draft versions of this thesis have been very helpful and the discussions we had have had a major influence on the final form and contents of this thesis.

Furthermore, I want to thank my colleagues at the University of Nijmegen, especially Marco Devillers, Maurice klein Gebbinck, John Kroeze, Mieke Massink, Frans Panken, Luc Rooijakkers, Marc Seutter and Rien Stein for offering such a pleasant environment to work in, and for the many cheerful discussions during coffee and lunch breaks.

I would like to thank ICT Automatisering Eindhoven BV and Paul Jansen for giving me the opportunity to finish this thesis during the time that I worked for Philips Natlab.

And finally, I thank my parents for always being there for me and for giving me a place where it is very easy to *not* think about my work.

Contents

Chapter 1

Introduction

1.1 Problem statement

The application of digital technology in all sorts of systems still is increasing. More and more analog electronic components are replaced by digital components. Initially this change of technology was only used to improve certain aspects of existing systems, such as production cost, reliability, processing speed and physical aspects such as size and weight. However, the functionality of those early digital systems basically was the same as the functionality of their analog predecessors. Nowadays, the integration of digital components has advanced so far that many modern systems are equipped with digital general purpose processors, together with so called *in product software* to tailor the processor for its specific task in the system. The presence of a general purpose processor, now makes it possible to add extra functionality to the system by implementing more features in the software.

This technological development does have major implications on the theory needed for designing systems. The first use of digital components gave rise to *discrete time* models. In discrete time models, time is divided in time slots of the same length, and changes in the state of the system can only take place on the boundaries of the time slots. Discrete time models are a specialization of *continuous time* models, in which the state of the system can change at any point in time. Analog systems are described using continuous time models Both discrete and continuous time models have been described adequately using theories expressed in classical mathematics. A well known example of mathematics used in this area is the theory of Fourier transforms and its application in digital signal processing.

With the use of integrated general purpose processors, the software design has become an important part of the design of a system. Many logics exist to reason about the correctness of programs. However, the digital and analog components of a system may interact in a non-trivial way. To show the correctness of such a system, we therefore need to be able to model the interface between those heterogeneous components, and to reason about the interaction taking place at the interface. In this respect, the programming languages and logics are deficient. The first because they do not provide constructs to describe hardware

and the latter because they do not support the mathematics needed to model hardware.

1.2 Funmath

Funmath (*Fun*ctional *math*ematics) is a specification language whose origin lies in the hardware world and which does not have these shortcomings. One basic design principle of the language is that it must allow to describe the the structure of a system without assigning a meaning to the structure. The subset of Funmath dealing with structure description is called *Reals* (*Real*izable *s*ystems). After the structure of a system has been described, various interpretations of the system can be obtained by applying the corresponding semantic functions to the structure description. This principle of multiple interpretations for a single structure description to deal with different aspects of a system, is called *system semantics*. Each interpretation reflects some mathematical model relevant to the analysis or design.

The executable part of Funmath is called *Comma* (*Com*putational *ma*thematics), and forms a functional programming language. Reals and Comma both are *operational* parts of Funmath. A language is called operational, if it only allows operational descriptions. A description is operational, if it can be mapped directly on a realization for the system described. Funmath also allows mathematically meaningful descriptions that are not operational. These descriptions are called *declarative*. A language allowing such descriptions, is also called declarative.

Declarativity is an essential requirement for system design languages: it must be possible to define a system by specifying its desired behaviour, which may result in a declarative system description. The aim of the design process then is to transform this declarative description into an operational description of a system having the same behaviour. This process can be performed completely within the Funmath language.

Funmath is designed so that its notation subsumes most mathematical notations in a natural way. It should not be seen as yet another new formalism, but as a restructuring of common mathematical notation, yielding new elements in the process. This makes Funmath a suitable language for theory development. Existing mathematical theories can be written in Funmath in a familiar way, and used in or related to new theories. This enables incremental theory development, instead of having to start from scratch for each development.

1.3 Relation to other work

Related general purpose specification languages are Z [42], Veritas [25] and Nuprl [29]. The main difference between these languages and Funmath is that the structure of Funmath is considerably simpler. There are less syntactic constructs, but these constructs are very orthogonal, so that constructs of other languages can be expressed as simple combinations of Funmath constructs.

The Funmath subset Reals can be compared to hardware description languages like VHDL [28], although in VHDL the separation between structure and behaviour is not as clear as in Reals. An environment for a precursor of Reals, named Glass (*G*eneral *la*nguage for *s*ystem *s*emantics) has been defined and implemented [41].

Comma is a functional programming language similar to Miranda [44] and Haskell [26]. As of yet, there is no programming environment available for Comma, but the translation of Comma to the implemented functional language Clean [22] has been investigated in [27].

In recent years, theories for verifying *hybrid systems*, (i.e. systems combining discrete and continuous components in a non-trivial way) have been developed based on State Automata (a form of discrete system theory). These models extend the discrete models with more refined time models and more general value spaces, while retaining as much as possible the (automatic) verification possibilities of the State Automata. This has resulted in various forms of Timed Automata [34, 2, 33]. These theories can be described in Funmath, but there only is limited (automatic) proof support implemented for Funmath [7] which is insufficient for these applications. However, Funmath does provide a framework in which such theories can be compared.

1.4 Aim of this dissertation

This dissertation focuses on two aspects of the full Funmath language: its notation and typing system. Both aspects play an important role in using Funmath for theory development. The notation must be flexible so that the theories can be written down in an easily recognizable way. The typing system must support various forms of polymorphism, that are needed be able to define general mathematical concepts in such way that they can be reused in different theories.

We will also demonstrate the suitability of Funmath as a specification language and as a language for theory development in the area of computing science. We will use Funmath to define its own type theory and to define a method for describing formal languages in Funmath. This method will be used to describe Funmath's own syntax. Funmath will also be used to specify algorithms to parse Funmath texts. Because these specifications are executable, they are part of the Comma subset of Funmath.

1.5 Organization of this dissertation

Chapter 2 introduces the Funmath language. The language allows various sorts of user-defined operators and includes a very general operator precedence mechanism for infix operators. Furthermore, some basic types and operators are given. The operators include type operators to construct function types such as arrow types and dependent types.

Chapter 3 shows how various forms of polymorphic typing are defined and used in Funmath. This chapter also is an example of theory development in Funmath. A theory of polymorphic functions is defined and type operators suited for typing polymorphic func-

tions are given. These type operators are variants of the function type operators given in Chapter 2, which in combination with type intersection can be used to type overloaded and implicitly polymorphic functions.

Chapter 4 defines some general mathematical concepts that are frequently used in Funmath, including familiar notions from domain theory and algebra. Furthermore, this chapter also defines some concepts often used in computing science, such as lists, direct extension and pattern matching, in a more general mathematical way, so that these concepts also can be used in other application areas than programming.

Chapter 5 defines a theory of *language functions* with which the structure and meaning of formal languages can be described in Funmath. The basic operators of the theory are language union and language concatenation. With these basic operators context-free languages can be described. The theory uses the direct extension of language union and pattern matching to describe more complicated languages including context-sensitive and even ambiguous languages. The theory is compared to the attribute and affix grammar formalisms, and to the theory of describing the meaning of language by semantic functions. Finally, several ways to resolve ambiguities in language functions are demonstrated.

The language function theory is used in Chapter 6 to give a lexical syntax and a formal grammar for Funmath. The grammar includes the features that make the notation so flexible: user-defined operators and operator precedence.

Chapter 7 gives some algorithms with which Funmath texts can be parsed efficiently. The crucial parsing algorithm, which uses the operator precedence relations to parse infix expressions deterministically, is proved correct.

Chapter 2

The Funmath language

This chapter gives an introduction to the formalism Funmath. A formalism consists of two parts: a *notation* and a *style of reasoning*. The notation of a formalism is characterized by its expressive power, determining the clarity, accuracy and the range of concepts expressible in the formalism, and the manipulative power, determining the degree to which it supports formal manipulation and the intended style of reasoning. Although the subject of this thesis is the formal definition of Funmath, parts of this chapter will have an informal character. This is done because the formal description of Funmath given in the following chapters is written in Funmath itself as an example of its expressiveness. This chapter gives an initial description of the formalism that will be elaborated in the chapters following.

2.1 Introduction

Funmath evolved over several years as a declarative formalism for the analysis, specification and design of hybrid systems. It was developed by Boute [15] and complemented by other contributions described later in this thesis. The need for a wide-spectrum declarative formalism became clear during a project for implementing a systems description environment for system semantics some ten years ago reported in [13], which also investigated modeling combined analog/digital systems. The name Funmath (*Fun*ctional *math*ematics) was introduced in 1989 and the formalism has only marginally changed since that time.

Another application area of Funmath is the formulation of new theories. An important feature of Funmath is that its notation is close to common mathematical notation, which makes it possible to write down mathematical theories in an easily recognizable way in a consistent syntactic framework. Examples of theory development using Funmath can be found in [43, 41, 35].

2.2 Funmath notation

2.2.1 Bindings

This topic may appear technical, but will be seen to be very useful in supporting and unifying the commonly used notational conventions in mathematics. In Funmath, *bindings* are used to introduce *identifiers* for objects. The basic form of a binding is

$$x : X \textbf{ with } P$$

where x is the indentifier for the object, X stands for expression denoting the *type* of the object, which, roughly speaking, is a set of which the object is a member, and P is an expression denoting the so called *filtering proposition* which the object must satisfy. The syntactic structure of expressions will be introduced in Section 2.2.3. An occurrence of an identifier in an expression is called *free* if it is not bound by a binding in that expression. Free occurrences of the identifier x in the filtering proposition P are bound by the binding, but free occurrences of x in the type X are not. The part $x : X$ of the binding, containing one identifier and it's type, is called a *binder*. Funmath also offers binders of the form $x := E$ which binds the identifier x to the object denoted by the expression E directly.

It is also possible to introduce a (hierarchical) tuple of identifiers for objects. For this purpose, Funmath has two tupling operators for bindings: the comma symbol is used for tupling identifiers, and the semicolon symbol is used for tupling binders. The semicolon symbol has precedence over **with** , so in the binding $x : X; y : Y$ **with** P, the filtering proposition P can be used to give properties of both x and y. Funmath also has the filtering operator $\wedge$ which has precedence over the semicolon, so that P in $x : X; y : Y \wedge P$ only belongs to the binder for y. Of course, it is also possible to use parentheses in bindings to override the predefined precedence. A complete binding syntax will be given in Chapter 6.

In the examples above, we only used simple alphabetic identifiers for objects. The range of identifiers which may be used is much wider, though: Funmath also allows special operator symbols, identifiers composed of more than one symbol, and the indication of argument positions by so called *argument placeholders*. If an identifier denotes a function we call it an *operator*. The notation of an operator is specified by its so called *operator pattern*, which is a sequence of operator symbols and placeholder symbols. There are two kinds of operators, each having their own placeholder symbol:

- *Regular operators:* these operators have a fixed number of arguments. The operator pattern of a regular operator is a sequence of the operator symbols of the operator with the placeholder symbol — on each argument position. Examples of such patterns are — + — for the infix operator + and [—] for the exfix operator [ ].

- *Variadic operators:* these operators can have any number of arguments greater than one, but only have one operator symbol. The operator pattern of a variadic operator consists of its operator symbol, followed by the placeholder symbol ....

There are some restrictions on the usage of the same operator symbol in different patterns, and also on the usage of argument placeholders. These restrictions are necessary to avoid ambiguities and will be presented in Chapter 6.

2.2.2 Declarations

Definitions

Definitions are used to introduce identifiers with a *global* scope. A definition consists of the keyword **def** followed by a binding. The basic form of a definition therefore is:

$$\textbf{def } x : X \textbf{ with } P$$

A definition is correct if and only if there is exactly *one* object x in the type denoted by X satisfying the proposition denoted by P. In the context of a definition, the filtering proposition P is called a *defining proposition*. The type X serves as a first approximation of the object, while the defining proposition P narrows this down to one individual object. The existence and uniqueness of the object are proof obligations for the definer. Identifiers introduced by definitions are called *constants* and have a global scope.

Specifications

Beside definitions, we also offer *specifications* to introduce identifiers for objects. The difference between specifications and definitions is that specifications do not have to satisfy the uniqueness requirement. The existence requirement must still be fulfilled, though. Specifications use the keyword **spec** instead of **def** :

$$\textbf{spec } n : \mathbb{Z} \textbf{ with } n < 10$$

This introduces an object n, of which we only know that it is an integer number smaller than 10. Using properties of n which cannot be derived from its specification, is unsafe. The reason for this is that we allow *strengthening* of specified objects, by simply giving more specifications for that object. For instance, we can strengthen the specification of n by saying

$$\textbf{spec } n : \mathbb{Z} \textbf{ with } n > -10$$

after which we know that n is an integer (the type $\mathbb{Z}$ containing all integers will be introduced in Section 2.3.2) satisfying $n < 10 \wedge n > -10$. It is also possible to give different types in different specifications for the same object. The type of the object then is the intersection of the different types. The existence requirement demands that this intersection may not be empty, though. For instance, it is allowed to strengthen the specification of n by **spec** $n : \mathbb{N}$, because the natural numbers are a subset of the integers.

2.2.3 Expressions

The Funmath expression syntax is the orthogonal combination of only *four* syntactic constructs which are in order of decreasing precedence: identifier, application, tuple and abstraction.

Identifier

An identifier denotes an object by name. Besides the identifiers introduced by bindings, Funmath also has the usual predefined identifiers for numbers and ASCII characters and strings. Character identifiers are surrounded by single quotes, e.g. '`a`' and text strings are surrounded by double quotes, e.g. "`hello`". Predefined identifiers may not be introduced as new identifiers in bindings, so there is no way to bind other objects to these identifiers.

Placeholders of operator identifiers may be omitted if the operator is used in an expression. This means that placeholder symbols are only needed when the operator is introduced in the binding.

Application

An application characterizes an object as the image of a given object under a given function. For example, if *double* is a function that doubles numbers then *double* 3 denotes the object 6. As can be seen, default function application is denoted by juxtaposition. It associates to the left, which means that $f\ a\ b$ should be read as $(f\ a)\ b$.

For operators that have a pattern with argument placeholders, like $— + —$, applications are written by substituting the arguments for the placeholders from left to right. So $3 + 5$ is an application of the operator $— + —$ to the argument pair $3, 5$. Applications of variadic operators, like $\times ...$, are written by alternating the arguments with the operator symbol, resulting in operator applications like $A \times B$ and $A \times B \times C$. This kind of application is called *operator application*. Default function application has precedence over operator application, so *double* $3 + 5$ means the same as (*double* 3) $+ 5$. Note that this implies that *prefix* operators like $\neg —$ also have precedence over default application, so that $\neg P\ x$ should be read as $\neg(P\ x)$. This shows that it does make a difference whether one uses the prefix pattern $f —$ or the simple pattern f for an operator.

The result of a function application may be undefined[1]. We denote all undefined applications with the symbol $\perp$. If the function part of an application is not a function, then the result is also undefined, e.g. $3\ 3 = \perp$. The *domain* of a function f is the collection of objects x for which the application $f\ x$ is defined. The *range* of a function is the collection

[1] Following a suggestion of the promotor to increase the flexibility of the type system in order to obtain a closer correspondence with standard mathematical practice, I have decided to exploit the possibilities of an explicit "undefined value" (which was already introduced in the functional formulation of the conditional [12]) throughout the entire formalism. Since this decision was taken at a rather late stage (i.e. near the completion of the manuscript) and does not pertain to the central theme of this thesis, not all ramifications regarding the axiomatization and style of reasoning have been thoroughly explored, and hence constitute an interesting topic for further study.

of images of all objects of the domain. A collection of objects may be called the *codomain* of a function if the range of the function is a subset of that collection. A consequence of this nomenclature is that each function has a unique domain and range, but may have many codomains. A more formal treatment of the notions domain and range will be given in Section 2.3.5.

Tuple

A tuple denotes a function whose domain is the set of index values of the tuple and whose mapping maps each index to the corresponding component of the tuple. For example, the tuple $2, 3, 5$ denotes a function whose domain is the set of its index values $\{0, 1, 2\}$ and whose mapping is given by $(2, 3, 5)\ 0 = 2$, $(2, 3, 5)\ 1 = 3$ and $(2, 3, 5)\ 2 = 5$.

Singleton tuples are denoted with the *tuple injection* operator τ. The application $\tau\ 3$ denotes a function whose domain contains only 0 and whose mapping is given by $\tau\ 3\ 0 = 3$. The constant ε denotes the empty function, that is, the domain of ε is the empty set $\emptyset$. As tuples represent functions, ε also serves as the empty tuple. Using $\perp$, we can define that $\tau\ x = (x, \perp)$ and $\varepsilon = (\perp, \perp)$. More elegant formal definitions for these operators will be given later.

Abstraction

An abstraction denotes an anonymous function. The basic form of abstraction is a binding followed by a dot and an expression:

$$x : X \mathbin{\mathbb{A}} P\ .\ E$$

The identifiers introduced in the binding are called the *variables* of the abstraction. Free occurrences of the variables in the *abstraction body* E, are bound by the binding. The abstraction denotes the function which maps all objects x of type X satisfying P to E. For abstractions, we have a variant of the β-reduction rule from λ-calculus [4]:

$$(x : X \mathbin{\mathbb{A}} P\ .\ E)\ F = E\ [F/x]$$

provided that F is a member of X satisfying $P\ [F/x]$, where — [—/—] is a syntactic substitution operator whose arguments are two expressions and an identifier, and which returns the first expression with all free occurrences of the given identifier replaced by the second expression argument. If F is not a member of X or doesn't satisfy $P\ [F/x]$, then the application is undefined:

$$(x : X \mathbin{\mathbb{A}} P\ .\ E)\ F = \perp$$

Therefore, the domain of the abstraction is a part of the collection of members of X satisfying P. If the abstraction is defined for *all* members of X satisfying P, we call it a *total* abstraction. Otherwise, the abstraction is called *partial*.

There are two alternative notations for abstractions in which the order of the components of the abstraction is rearranged. The meaning of these notations is given by:

$$(E \mid x : X \mathbin{\&} P) = (x : X \mathbin{\&} P \;.\; E)$$
$$(x : X \mid P) = (x : X \mathbin{\&} P \;.\; x)$$

These abstraction notations using the symbol | instead of symbol . are due to Van Thienen [43]. They were introduced to be able to write set comprehension in the familiar way, as will be shown in Section 2.3.5.

Abstractions with tupled bindings denote functions on tuples:

$$(x : \mathbb{N}; y : \mathbb{Z} \;.\; x + y) = (x, y : \mathbb{N} \times \mathbb{Z} \;.\; x + y) = (z : \mathbb{N} \times \mathbb{Z} \;.\; z\ 0 + z\ 1)$$

Note that the above abstractions are different from the nested abstraction $x : \mathbb{N} \;.\; y : \mathbb{Z} \;.\; x + y$ which takes arguments one at a time, whereas the previous abstractions take the two arguments together in a tuple.

2.2.4 Partial applications

Besides tuples and abstractions, Funmath offers yet another way to denote functions: *partial applications*. A partial application is an operator application in which one or more arguments are missing. In functional programming languages, partial applications are often called *sections*. A partial application denotes a function which takes the missing arguments and substitutes them from left to right in the empty spots in the operator application. For instance the partial application $(3+)$ of the operator $— + —$, is the function that takes an object x and maps it to $3 + x$, so we have that $(3+)\ x = 3 + x$. It is also possible to leave out more than one argument. The result is a function that takes a tuple of arguments, for instance $(+)\ (x, y) = x + y$. This also shows that the operator name itself can be seen as a partial application in which all arguments are missing.

Partial applications can easily be transformed to equivalent abstractions, by introducing new identifiers for the missing arguments, and binding these identifiers to the universal type $\mathcal{U}$, which contains all objects including $\bot$. For instance, $(3+) = (x : \mathcal{U} \;.\; 3 + x)$ and $(+) = (x : \mathcal{U}; y : \mathcal{U} \;.\; x + y)$. Abstractions obtained this way usually are partial.

2.2.5 Operator precedence declarations

Definitions and specifications are semantic declarations, because, besides introducing names, they also give meaning to the names. The precedence declarations described below are purely syntactic: they only specify syntactic precedence between infix and variadic operators. Strictly speaking, there is no need for operator precedence declarations, because one always can use parentheses to surround arguments of operators. In practice, however, the absence of an operator precedence mechanism soon leads to unwieldy expressions.

Operator precedence can be defined between any pair of operators having arguments on both their left hand and right hand side, which are exactly the infix operators and variadic operators. Precedence declarations start with the keyword **par** , followed by a *parenthesis insertion pattern*, which we also will call a *precedence pattern*. A precedence pattern is an operator application of infix or variadic operators, in which all operator arguments

are surrounded by parentheses. The effect of such pattern is that applications *without* the parenthesis are parsed as indicated by the pattern. For instance, the precedence declaration

par $(— \cdot —) + —$

specifies that $1 \cdot 2 + 3$ has to be parsed as $(1 \cdot 2) + 3$. Precedence patterns can also have arguments on both sides, as in

par $(— \cdot —) + (— \cdot —)$

which has the same effect as the following two declarations:

par $(— \cdot —) + —$
par $— + (— \cdot —)$

Nesting is also allowed in precedence patterns. For instance,

par $((— - —) + —) - —$

abbreviates

par $(— + —) - —$
par $(— - —) + —$

Furthermore, operator precedence is transitive, which means that outer arguments of a left argument operator of an operator *op*, also are valid left arguments of *op*. Similarly, all outer arguments of a right argument operator of *op* are also valid right arguments of *op*. This means that the two declarations above also infer the precedence patterns $(— + —) + —$ and $(— - —) - —$. The transitivity rules for operator precedence will be defined formally in Chapter 6. Patterns which specify precedence between different occurrences of a single operator are also allowed. These patterns are used to specify that an infix operator associates to the left or to the right. For instance, to specify that the function arrow $\to$ associates to the right we write:

par $— \to (— \to —)$

Note that because this operator only has one operator symbol, it is also possible to define it as a variadic operator $\to \ldots$ having the *semantic* property $(a \to b \to c) = (a \to (b \to c))$.

The scope of precedence declarations is global, and has no relation with the scope of identifiers; even if an operator pattern is locally bound to another object, its syntactic precedence will be preserved. As with operator patterns, there are some restrictions to prevent ambiguities, which can be found in Chapter 6. Appendix A lists the precedence of all operators used in this manuscript.

The syntactic precedence between operators which are not both infix or variadic operators is predefined: prefix application has precedence over postfix application, which in turn has precedence over infix and variadic application.

2.3 Primitive types and operators

In this section, we introduce some basic types and operators on those types. We do not intend to give a complete axiomatization of these types and operators, but in most cases we supply definitions or specifications that provide sufficient understanding of the operators, and which can be used in proofs about objects defined in terms of the primitive operators. Funmath should be seen as a notational framework, in which theories can be defined by specifying the notation of the operators of the theory, and the properties of the operators, using definitions and specifications. Note that at this stage, we do not yet have function type operators (these will be introduced in Section 2.5), so the types used in this section cannot give domain information about the operators introduced. This lack of domain information is also the reason that most operators are introduced by specifications instead of definitions, which require the domain to be defined uniquely. The syntactic precedence of the operators introduced in this section is given in Appendix A.

2.3.1 Logic

The basic version of Funmath supports the classical view on logic: there are two truth values, 0 for false and 1 for true, which form the contents of the type $\mathbb{B}$. The logical connectives $— \wedge —$, $— \vee —$, $— \Rightarrow —$, $— \Leftarrow —$, $— \equiv —$ (for logical equivalence) and the negation operator $\neg —$ are defined on the truth values in the usual way. Most of these operators are not strict. That is, if one of the arguments is undefined, then the result can still be defined. We have the following non-strictness properties:

$$(0 \wedge \bot) = (\bot \wedge 0) = 0$$
$$(1 \vee \bot) = (\bot \vee 1) = 1$$
$$(0 \Rightarrow \bot) = (\bot \Rightarrow 1) = 1$$

All other applications of these operators involving $\bot$ yield $\bot$. This corresponds to the three valued logic used by Kleene [30]. A complete axiomization of this logic can be found in [6], which also discusses other three valued logics. The non-strictness properties of the Funmath logic enable us to write propositions like $x \neq 0 \Rightarrow (f\ x = 1/x)$, which otherwise would have been undefined for $x = 0$.

Funmath also provides the *universal quantifier* $\forall —$ and the *existential quantifier* $\exists —$. Both quantifiers are higher order functions, mapping predicates (i.e. functions with codomain $\mathbb{B}$) to truth values. The argument of $\forall$ is a predicate P yielding truth values and the application $\forall P$ indicates whether the range of P contains at most the truth value 1, i.e. the range is either empty or it is the singleton set containing 1. Similarly, $\exists$ is defined so that $\exists P$ holds if P has 1 in its range. Now there is no need for universal and existential quantification as separate syntactic language constructs, because we can write quantifications by applying the corresponding quantifier to an abstraction with body of type $\mathbb{B}$. For example, the proposition $\exists(n : \mathbb{N} \,.\, n = 3)$ is true, because the abstraction $n : \mathbb{N} \,.\, n = 3$ maps 3 to the truth value 1. The proposition $\forall(n : \mathbb{N} \,.\, n = n)$ is also true, because every $n : \mathbb{N}$ is mapped by $n : \mathbb{N} \,.\, n = n$ to the truth value 1. Note that, when applied to

pairs of truth values, the quantifiers behave the same as the corresponding connective, e.g. $\forall(a, b) = (a \wedge b)$.

2.3.2 Arithmetic

For arithmetic, Funmath supports the usual types $\mathbb{N}$ containing the natural numbers, $\mathbb{Z}$ containing the whole numbers, $\mathbb{Q}$ containing the rational numbers, $\mathbb{R}$ containing the real numbers and $\mathbb{C}$ containing the complex numbers, together with the usual arithmetic operators, like $— + —$, $— - —$, $— \cdot —$, $—/—$, $\overline{—}$ and $\sqrt{—}$. Not all types are closed under all operators. We have, for instance, that $2 \in \mathbb{N} \wedge 3 \in \mathbb{N}$, but not $2 - 3 \in \mathbb{N}$. Undefinedness is represented by $\bot$. For instance, $1/0 = \bot$, Furthermore, the equality relation $= ...$ is defined for arithmetic values and the undefined object $\bot$ is not contained in any of the arithmetic types, so $\mathbb{R}$ is not closed under division, but $\mathbb{R}$ augmented with $\bot$ is.

2.3.3 Conditional operators

For conditional expression, Funmath offers the *guard* operator $— \,?\, —$ and the *else* operator $— \,\dagger\, —$, which were introduced in [12]. The guard operator takes a condition and a value. If the condition is true, it yields the given value. Otherwise, the result is undefined:

$$\textbf{spec}\ (— \,?\, —) : \mathcal{U}\ \textbf{with}\ \forall(a : \mathcal{U}\ .\ (0 \,?\, a) = \bot \wedge (1 \,?\, a) = a)$$

where $\mathcal{U}$ is the universal type containing all Funmath objects, including $\bot$. The else operator has two arguments. If the first argument is defined, then it is returned. Otherwise, the second argument is returned:

$$\textbf{spec}\ (— \,\dagger\, —) : \mathcal{U}\ \textbf{with}\ \forall(a : \mathcal{U} \setminus \{\bot\};\ b : \mathcal{U}\ .\ (\bot \,\dagger\, b) = b \wedge (a \,\dagger\, b) = a)$$

2.3.4 Set operations

Funmath uses the *type universe* $\mathcal{T}$ as the type of all types. The undefined object $\bot$ is not a member of $\mathcal{T}$. The *type membership* relation $— \in —$, its negation $— \notin —$, and the *subset* relation $— \subseteq —$ satisfy:

$$\textbf{spec}\ (— \in —) : \mathcal{U}$$
$$\textbf{with}\ \forall(x : \mathcal{U}; A : \mathcal{T}\ .\ (x \in A) \equiv \exists(y : A\ .\ x = y))$$

$$\textbf{spec}\ (— \notin —) : \mathcal{U}$$
$$\textbf{with}\ \forall(x : \mathcal{U}; A : \mathcal{T}\ .\ (x \notin A) \equiv \neg(x \in A))$$

$$\textbf{spec}\ (— \subseteq —) : \mathcal{U}$$
$$\textbf{with}\ \forall(A : \mathcal{T}; B : \mathcal{T}\ .\ (A \subseteq B) \equiv \forall(x : A\ .\ x \in B))$$

Equality on sets is extensional:

spec $(= ...) : \mathcal{U}$
with $\forall(A : \mathcal{T}; B : \mathcal{T} \;.\; A = B \equiv \forall(x : \mathcal{U} \;.\; x \in A \equiv x \in B))$

We now introduce a type operator *Fam*, so that $Fam\ A$ is the type containing all *families* of (elements of) A, i.e. all functions whose range is a subet of A:

def $Fam : \mathcal{U}$
with $\mathcal{D}\ Fam = \mathcal{T} \wedge \forall(f : \mathcal{U}; A : \mathcal{T} \;.\; f \in Fam\ A \equiv f \in \mathcal{F} \wedge \{f\} \subseteq A)$

For $f : Fam\ A$ with $\mathcal{D}\ f = B$ we also call f a B-indexed family of A. In Funmath the notion of *family* as commonly used in mathematics, physics and engineering in a rather unstructured way is subsumed by the notion of *function* by simply considering both terms as synonyms. The term *family of* A is used for designating a function with codomain A whenever its domain (in this nomenclature called *index set*) may be chosen rather arbitrarily or is not of particular interest to the discussion.

Using *Fam* we can define the *type union* operator $\bigcup$ and the *type intersection* operator $\bigcap$, which both operate on families of types:

def $(\bigcup —) : \mathcal{U}$
with $\mathcal{D} \bigcup = Fam\ \mathcal{T} \wedge \forall(x : \mathcal{U}; F : Fam\ \mathcal{T} \;.\; x \in \bigcup F \equiv \exists(A : \{F\} \;.\; x \in A))$

def $(\bigcap —) : \mathcal{U}$
with $\mathcal{D} \bigcap = Fam\ \mathcal{T} \wedge \forall(x : \mathcal{U}; F : Fam\ \mathcal{T} \;.\; x \in \bigcap F \equiv \forall(A : \{F\} \;.\; x \in A))$

We also have the variadic versions $\cup...$ and $\cap...$ which are only defined on tuples of types, the empty set $\emptyset$ and the *set difference* operator $— \setminus —$, which only operates on pairs of types.

def $\emptyset : \mathcal{T}$ **with** $\forall(x : \mathcal{U} \;.\; x \notin \emptyset)$
spec $(— \setminus —) : \mathcal{U}$
with $\forall(A : \mathcal{T}; B : \mathcal{T}; x : \mathcal{U} \;.\; x \in (A \setminus B) \equiv x \in A \wedge x \notin B)$

Because Funmath uses the notation $\{—\}$ to denote the range of a function, this notation cannot be used to denote singleton sets. For this purpose Funmath defines the *set injection* operator ι, which maps an object to the singleton set containing that object:

def $(\iota —) : \mathcal{U}$
with $\mathcal{D}\ \iota = \mathcal{U} \wedge \forall(x : \mathcal{U}; y : \mathcal{U} \;.\; x \in \iota\ y \equiv x = y)$

Using ι for singleton sets instead of curly braces is not unusual in mathematics (see [23]).

Finally, the *power set* operator $\mathcal{P}$ is given by:

def $(\mathcal{P} —) : \mathcal{U}$
with $\mathcal{D}\ \mathcal{P} = \mathcal{T} \wedge \forall(A : \mathcal{U}; B : \mathcal{T} \;.\; A \in \mathcal{P}\ B \equiv A \in \mathcal{T} \wedge A \subseteq B)$

2.3.5 Domain and range of functions

In Funmath, we postulate a *function universe* $\mathcal{F}$, which is the type of all functions. The undefined object $\bot$ is not considered to be a function and therfore *not* contained in $\mathcal{F}$. Note that in Funmath functions are not considered as a special case of sets or relations. Sets and functions are different objects, i.e. $\mathcal{T} \cap \mathcal{F} = \emptyset$. In Section 4.5 we show that in Funmath relations even are a special case of functions on booleans.

On functions, we have the *domain* operator $\mathcal{D}$ — and the range operator $\{—\}$. The domain of a function is the collection of values for which the function is defined:

spec $(\mathcal{D}\ —) : \mathcal{F}$
with $\forall(f : \mathcal{F}; x : \mathcal{U}\ .\ x \in \mathcal{D}\ f \equiv f\ x \neq \bot)$

Function equality is, like set equality, extensional, which is specified by:

spec $(= ...) : \mathcal{F}$
with $\forall(f : \mathcal{F}; g : \mathcal{F}\ .\ f = g \equiv \forall(x : \mathcal{U}\ .\ f\ x = g\ x))$

A consequence of the definition of function equality and conditionals, and the semantics of abstraction is that filtering propositions and even type information of abstractions can be moved from the bindings to the abstraction bodies:

$$(x : X \mathbin{\&} P\ .\ E) = (x : X\ .\ P\ ?\ E)$$
$$(x : X \mathbin{\&} P\ .\ E) = (x : \mathcal{U} \mathbin{\&} x \in X \wedge P\ .\ E) = (x : \mathcal{U}\ .\ (x \in X \wedge P)\ ?\ E)$$

The range of a function is the collection of values for which there is a domain value which the function maps to the range value:

spec $(\{—\}) : \mathcal{F}$
with $\forall(f : \mathcal{F}; y : \mathcal{U}\ .\ y \in \{f\} \equiv \exists(x : \mathcal{D}\ f\ .\ f\ x = y))$

Note that it is, by definition, not possible for a function to have $\bot$ in its range. The range operator combines nicely with Van Thienen notation for abstraction to set comprehension. For instance, $\{2 \cdot x \mid x : \mathbb{Z}\}$ denotes the range of the function $2 \cdot x \mid x : \mathbb{Z}$, which is an alternative notation for $x : \mathbb{Z}\ .\ 2 \cdot x$. The range of this function contains exactly all even integers, which is precisely what the expression $\{2 \cdot x \mid x : \mathbb{Z}\}$ suggests. Another example is $\{x : \mathbb{Z} \mid x > 10\}$, which denotes the range of the function $x : \mathbb{Z} \mid x > 10$ which is an abbreviation of $x : \mathbb{Z} \mathbin{\&} x > 10\ .\ x$. This is the identity function on the set of integers larger than 10. So, as suggested by the notation, the range of this function equals its domain, the set of integers larger than 10.

Note that for every partial abstraction we can always find an equivalent total abstraction by adding explicit type correctness conditions to the filtering proposition of the partial abstraction for every function application that may yield undefined. For instance, the partial abstraction $x : \mathbb{R}\ .\ 1/x$ is made total by adding the type correctness condition $(1, x) \in \mathcal{D}\ (/)$ which results in the total abstraction $x : \mathbb{R} \mathbin{\&} (1, x) \in \mathcal{D}\ (/)\ .\ 1/x$. Another example is

the *function composition* of functions f and g, which is given by the partial abstraction $x : \mathcal{D}\ g\ .\ f\ (g\ x)$. A total version of this abstraction is $x : \mathcal{D}\ g \wedge g\ x \in \mathcal{D}\ f\ .\ f\ (g\ x)$.

For tuples, we use the same terminology as for abstractions: the domain of a tuple is the set of the index values of the defined components. If all components are defined, we call the tuple *total* and if one or more of the components are undefined we call it *partial*. In contrast to abstractions, not every partial tuple has a total equivalent. The combination of the range operator with total tuples gives us the familiar notation of finite sets. The expression $\{a, b, c\}$ denotes the range of the tuple a, b, c. A tuple is a mapping which maps index values to the corresponding components of the tuple, so the range of a, b, c is exactly the set containing a, b and c.

2.4 Transformational reasoning

Funmath supports the *transformational proof style*. By the convention attributed to W. Feyen, transformational proofs have the following appearance:

$$\begin{array}{lll} E_0 & & \\ \quad R_0 & \{\ \mathit{justification}_0\} & E_1 \\ \quad R_1 & \{\ \mathit{justification}_1\} & E_2 \end{array}$$

where every R_i is a relation over the domain of the expressions E_i. The proposition $\mathit{justification}_i$ has to motivate the validity of the proposition $R_i(E_i, E_{i+1})$. Often the justification is the observation that this proposition is an instance of a definition or specification, or a theorem proved earlier. Furthermore, monotonicity properties of the relations R_i are often used silently. More detailed elaborations on the transformational proof style in Funmath can be found in [17, 43, 35, 7].

2.5 Function type operators

A *function type* is a type containing only functions. The type of all function types therefore is $\mathcal{P}\ \mathcal{F}$. One of the most important type constructors of Funmath is the *generalized Cartesian product* operator $\bigtimes$, which maps a family F of types to the function type $\bigtimes F$, containing all functions f whose domain is a subset of the domain of F and mapping each argument $x : \mathcal{D}\ F$ to a member of the type $F\ x$:

$$\textbf{def}\ (\bigtimes —) := (F : \mathit{Fam}\ \mathcal{T}\ .\ \{f : \mathcal{F} \mid \mathcal{D}\ f \subseteq \mathcal{D}\ F \wedge \forall (x : \mathcal{D}\ F\ .\ f\ x \in F\ x)\})$$

The symbol $\bigtimes$ is used because it is a large version of the symbol $\times$, which already indicates its relation with Cartesian products: application of $\bigtimes$ to a tuple of types yields the Cartesian product of those types. For instance:

$$\begin{array}{ll} & \bigtimes(A, B) \\ = & \qquad \{\,\text{def}\ \bigtimes\,\} \end{array}$$

$$
\begin{array}{ll}
 & \{f : \mathcal{F} \mid \mathcal{D}\ f \subseteq \mathcal{D}\ (A,B) \wedge \forall(x : \mathcal{D}\ (A,B)\ .\ f\ x \in (A,B)\ x)\} \\
= & \quad \{\ \mathcal{D}\ (A,B) = \{0,1\}\ \} \\
 & \{f : \mathcal{F} \mid \mathcal{D}\ f \subseteq \{0,1\} \wedge \forall(x : \{0,1\}\ .\ f\ x \in (A,B)\ x)\} \\
= & \quad \{\ \mathcal{D}\ f \subseteq \{0,1\} \equiv \exists(a : \mathcal{U}; b : \mathcal{U}\ .\ f = (a,b))\ \} \\
 & \{a : \mathcal{U}; b : \mathcal{U} \mid \forall(x : \{0,1\}\ .\ (a,b)\ x \in (A,B)\ x)\} \\
= & \quad \{\ \forall \text{ and } \wedge \text{ behave the same on tuples}\ \} \\
 & \{a : \mathcal{U}; b : \mathcal{U} \mid (a,b)\ 0 \in (A,B)\ 0 \wedge (a,b)\ 1 \in (A,B)\ 1\} \\
= & \quad \{\ \text{tuple indexing}\ \} \\
 & \{a : \mathcal{U}; b : \mathcal{U} \mid a \in A \wedge b \in B\}
\end{array}
$$

Therefore, we can define $\times$ as a variadic version of $\bigtimes$ by

$$\textbf{def}\ (\times ...) := \bigtimes$$

so that $A \times B \times C$ stands for $\bigtimes(A,B,C)$. Note that because in the definition of $\bigtimes$ we wrote $\mathcal{D}\ f \subseteq \mathcal{D}\ F$ and not $\mathcal{D}\ f = \mathcal{D}\ F$, we can also denote partial products. We have, for instance, that:

$$A \times B \times (C \cup_\iota \bot) = (A \times B \times C) \cup (A \times B)$$

This shows that to indicate that the members of $\bigtimes F$ may be undefined for argument x, we just have to add $\bot$ to $F\ x$. On the other hand, by not putting $\bot$ in $F\ x$ we state that *all* members f of $\bigtimes F$ are defined for argument x, and therefore satisfy $x \in \mathcal{D}\ f$.

The function type $A \to B$ contains all functions whose domain is (a part of) A, and which map any argument from A to a member of B. Therefore, arrow types are a special case of product types, where the result type does not depend on the argument:

$$\textbf{def}\ (— \to —) := (A : \mathcal{T}; B : \mathcal{T}\ .\ \bigtimes(x : A\ .\ B))$$

For $f : A \to B$ with $\bot \notin B$ we have $\mathcal{D}\ f = A$. We use the following precedences for the function type operators:

$$\textbf{par}\ (— \to —) = (— \to —)$$
$$\textbf{par}\ (— \times —) \to (— \times —)$$
$$\textbf{par}\ (— \cup —) \times (— \cup —)$$

We showed that combining the generalized Cartesian product operator $\bigtimes$ with tuples of types yields Cartesian products. Combining $\bigtimes$ with abstraction gives us dependent types and parametric polymorphism. An example of this is the following definition of the parametrized identity function id':

$$\textbf{def}\ id' : \bigtimes(A : \mathcal{T}\ .\ A \to A)$$
$$\textbf{with}\ \forall(A : \mathcal{T}\ .\ id'\ A = (x : A\ .\ x))$$

Elaborating the type statement $id' \in \bigtimes(A : \mathcal{T} \;.\; A \to A)$ and using $\forall(A : \mathcal{T} \;.\; \bot \notin A \to A)$ yields

$$\mathcal{D}\ id' = \mathcal{T} \wedge \forall(A : \mathcal{T} \;.\; id'\ A \in A \to A)$$

So the product type made it possible to express that the type of $id'\ A$ is $A \to A$ which depends on A. This would not have been possible by using just arrow types.

Another example of the use of $\bigtimes$ in dependent types is the *deterministic choice* operator, with notation $[\![—]\!]$. This operator maps a nonempty function to a value in its range, which can be expressed using a product type:

$$\textbf{spec}\ [\![—]\!] : \bigtimes(f : \mathcal{F} \setminus \iota\ \varepsilon \;.\; \{f\})$$

Expanding the definition of $\bigtimes$ and using $\forall(f : \mathcal{F} \;.\; \bot \notin \{f\})$ yields:

$$\mathcal{D}[\![]\!] = \mathcal{F} \setminus \iota\ \varepsilon \wedge \forall(f : \mathcal{F} \setminus \iota\ \varepsilon \;.\; [\![f]\!] \in \{f\})$$

This states that the choice operator can be applied to any non-empty function and that the result of the application is an element of the range of that function. So if the range of the function f contains *one* element, then $[\![f]\!]$ must denote that element. In combination with Van Thienen notation for abstractions, this feature offers a convenient way to write local definitions. If we need a local definition of an object x of type X with defining property P, we can simply write $[\![x : X \mid P]\!]$. If the object x is used locally in an expression E, we can write $[\![E \mid x : X \mathbin{.\!\wedge} P]\!]$. Note that for functions containing more than one element, the choice is left unspecified. The only thing we know in this case is that the choice operator behaves as a proper function. That is, given the same function it returns the same range value. This is why we call this choice operator *deterministic*.

2.6 Operators on functions

In this section some general operators are defined for denoting and manipulating functions.

Constant functions are denoted by the *constant function definer* $^\bullet$ which takes the value that the constant function should yield as first argument and the domain of the function as second argument:

$$\textbf{def}\ (—^\bullet—) : \mathcal{U} \times \mathcal{T} \to \mathcal{F}$$
$$\textbf{with}\ \forall(x : \mathcal{U}; A : \mathcal{T} \;.\; (x^\bullet A) = (y : A \;.\; x))$$

$$\textbf{par}\ (—^\bullet—) = (—^\bullet—)$$
$$\textbf{par}\ —^\bullet(— \to —)$$

Funmath also uses the *constantness* predicate *con* which yields true iff the given function is constant:

def $con : \mathcal{F} \to \mathbb{B}$
with $\forall(f : \mathcal{F} \;.\; con\ f \equiv f \in \{\bullet\})$

It can be shown that the definition of *con* is equivalent to the familiar notion of constantness of a function:

$$\forall(f : \mathcal{F} \;.\; con\ f \equiv \exists(x : \mathcal{U} \;.\; \forall(y : \mathcal{D}\ f \;.\; f\ y = x)))$$

The existentially bound variable x can actually be substituted by $[\![f]\!]$:

$$\forall(f : \mathcal{F} \;.\; con\ f \equiv \forall(y : \mathcal{D}\ f \;.\; f\ y = [\![f]\!])) \tag{2.1}$$

This replacement is possible because if f is constant, then it is either empty in which case $\mathcal{D}\ f = \emptyset$ and the right-hand side of the equivalence is trivially true, or its range contains exactly one element which by the type of the deterministic choice operator $[\![]\!]$ must be $[\![f]\!]$. For nonempty constant functions it is therefore possible to define a choice operator uniquely by

def $([—]) : \bigtimes(f : \{\bullet\} \setminus \iota\ \varepsilon \;.\; \{f\})$

This operator is the restriction of the deterministic choice operator $[\![]\!]$ to constant functions. We call it the *safe choice* operator because it only makes a choice if there is only one possibility from which to choose. We extend $=$ to a variadic notation by specifying that for arguments that are no pair, equality is the same as constantness:

spec $(= ...) : \mathcal{F}$
with $\forall(F : Fam\ \mathbb{B} \wedge \neg(\mathcal{D}\ F \subseteq \mathbb{B}) \;.\; (=)\ F \equiv con\ F)$

Now we can write $x = y = z$ as an abbreviation for $con\ (x, y, z)$. Incidentally, this is the same as $x = y \wedge y = z$ in the case that x, y and z are all defined.

The *one-point function definer* $\mapsto$ is used to denote functions which map a given value to another given value:

def $(— \mapsto —) : \mathcal{U} \times \mathcal{U} \to \mathcal{F}$
with $\forall(x : \mathcal{U}; y : \mathcal{U} \;.\; (x \mapsto y) = y^{\bullet}\iota\ x)$

par $(— \mapsto —) = (— \mapsto —)$
par $(— \to —) \mapsto (— \to —)$

The generalized *function composition* operator composes any pair of functions into a new function:

def $(— \circ —) : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$
with $\forall(f : \mathcal{F}; g : \mathcal{F} \;.\; (f \circ g) = (x : \mathcal{D}\ g \;.\; f\ (g\ x)))$

par $(— \circ —) = (— \circ —)$

The domain of the composition $f \circ g$ contains all objects $x : \mathcal{D}\ g$ satisfying $g\ x \in \mathcal{D}\ f$, which are those objects for which the application of f to the result of g is defined. This makes it possible to compose *any* pair of functions, instead of just functions f and g satisfying $\{g\} \subseteq \mathcal{D}\ f$. The empty function ε and the universal identity function id are defined by

$$\textbf{def}\ \varepsilon : \mathcal{F}\ \textbf{with}\ \mathcal{D}\ \varepsilon = \emptyset$$
$$\textbf{def}\ id : \mathcal{F}\ \textbf{with}\ id = (x : \mathcal{U}\ .\ x)$$

The function id is the unit element of $\circ$ and ε is the zero element of $\circ$:

$$\forall(f : \mathcal{F}\ .\ f \circ id = id \circ f = f)$$
$$\forall(f : \mathcal{F}\ .\ f \circ \varepsilon = \varepsilon \circ f = \varepsilon)$$

Because tuples and abstractions denote functions, the function composition operator can be used to distribute functions over tuples and abstractions. If we have a function f, then $f \circ (a, b, c)$ equals $(f\ a, f\ b, f\ c)$ and $f \circ (x : X \wedge P\ x\ .\ E\ x)$ equals $(x : X \wedge P\ x\ .\ f\ (E\ x))$.

The *function domain restrictor* restricts the domain of a given function to a given set:

$$\textbf{def}\ (— \rceil —) : \mathcal{F} \times \mathcal{T} \to \mathcal{F}$$
$$\textbf{with}\ \forall(f : \mathcal{F}; A : \mathcal{T}\ .\ (f \rceil A) = (x : A\ .\ f\ x))$$

Note that the abstraction $x : A\ .\ f\ x$ is partial. An equivalent total version is $x : \mathcal{D}\ f \cap A\ .\ f\ x$.

$$\textbf{par}\ (— \rceil —) = (— \rceil —)$$
$$\textbf{par}\ (— \rceil —) \rceil —$$
$$\textbf{par}\ — \rceil (— \to —)$$

This operator has the following properties:

$$f \rceil \mathcal{D}\ f = f$$
$$f \rceil A \rceil B = f \rceil (A \cap B)$$

Another important operation on functions is taking the *inverse*. The function inverse operator is defined in a few steps. Funmath defines the *bijective domain* operator $\mathcal{B}$ and the *bijective range* operator $\mathcal{R}$:

$$\textbf{def}\ (\mathcal{B}\ —) : \mathcal{F} \to \mathcal{T}$$
$$\textbf{with}\ \forall(f : \mathcal{F}; x : \mathcal{U}\ .\ x \in \mathcal{B}\ f \equiv x \in \mathcal{D}\ f \wedge \forall(y : \mathcal{D}\ f\ .\ f\ x = f\ y \Rightarrow x = y))$$

$$\textbf{def}\ (\mathcal{R}\ —) : \mathcal{F} \to \mathcal{T}$$
$$\textbf{with}\ \forall(f : \mathcal{F}\ .\ \mathcal{R}\ f = \{f \rceil \mathcal{B}\ f\})$$

The bijective domain of a function is that part of its domain on which the function is injective. So a function is injective if its domain equals its bijective domain:

def $inj : \mathcal{F} \to \mathbb{B}$
with $\forall(f : \mathcal{F} \;.\; inj\ f \equiv \mathcal{B}\ f = \mathcal{D}\ f)$

In the same way that we introduced $=$ as the variadic notation for *con*, we now can define $\neq$ as the variadic notation for *inj*:

spec $(\neq \ldots) : \mathcal{F}$
with $\forall(F : Fam\ \mathbb{B} \wedge \neg(\mathcal{D}\ F \subseteq \mathbb{B}) \;.\; (\neq)\ F \equiv inj\ F)$

par $(— = —) \neq (— = —)$

By the transitivity of operator precedence, this precedence declaration specifies that all operators having precedence over $=$, also have precedence over $\neq$.

Instead of defining the inverse operator only for injective functions, we define it for every function by taking the inverse over the bijective part of the function:

def $(—^{-}) : \mathcal{F} \to \mathcal{F}$
with $\forall(f : \mathcal{F} \;.\; \mathcal{D}\ (f^{-}) = \mathcal{R}\ f \wedge \forall(x : \mathcal{B}\ f \;.\; f^{-}(f\ x) = x))$

So we have $f^{-} = (f \rceil \mathcal{B}\ f)^{-}$.

Using the function restrictor $\rceil$ we define the *subfunction* relation

def $(— \sqsubseteq —) : \mathcal{F} \times \mathcal{F} \to \mathbb{B}$
with $\forall(f : \mathcal{F}; g : \mathcal{F} \;.\; (f \sqsubseteq g) \equiv f = g \rceil \mathcal{D}\ f)$
par $(— \sqsubseteq —) \wedge (— \sqsubseteq —)$

which has the following properties:

$$f \rceil A \sqsubseteq f \tag{2.2}$$
$$f \sqsubseteq g \equiv \forall(x : \mathcal{D}\ f \;.\; f\ x = g\ x) \tag{2.3}$$

Proof of (2.3)

$$\begin{array}{ll} & f \sqsubseteq g \\ \equiv & \quad \{\text{def } \sqsubseteq\} \\ & f = g \rceil \mathcal{D}\ f \\ \equiv & \quad \{\text{function equality}\} \\ & \forall(x : \mathcal{U} \;.\; f\ x = (g \rceil \mathcal{D}\ f)\ x) \\ \equiv & \quad \{\text{def } \rceil\} \\ & \forall(x : \mathcal{U} \;.\; f\ x = (y : \mathcal{D}\ f \;.\; g\ y)\ x) \\ \equiv & \quad \{\beta\text{-reduction}\} \\ & \forall(x : \mathcal{U} \;.\; f\ x = (x \in \mathcal{D}\ f\ ?\ g\ x)) \\ \equiv & \quad \{\text{case distinction}\} \end{array}$$

$$
\begin{array}{ll}
 & \forall(x : \mathcal{D}\ f\ .\ f\ x = (x \in \mathcal{D}\ f\ ?\ g\ x)) \wedge \forall(x : \mathcal{U} \setminus \mathcal{D}\ f\ .\ f\ x = (x \in \mathcal{D}\ f\ ?\ g\ x)) \\
\equiv & \quad \{\,\text{def ?}\,\} \\
 & \forall(x : \mathcal{D}\ f\ .\ f\ x = g\ x) \wedge \forall(x : \mathcal{U} \setminus \mathcal{D}\ f\ .\ f\ x = \bot) \\
\equiv & \quad \{\,\text{spec } \mathcal{D}\,\} \\
 & \forall(x : \mathcal{D}\ f\ .\ f\ x = g\ x)
\end{array}
$$

So f is a subfunction of g if and only if f and g behave the same on the domain of f. From this we can also derive that the domain of f is a subset of the domain of g:

$$
\begin{array}{ll}
 & \forall(x : \mathcal{D}\ f\ .\ f\ x = g\ x) \\
\Rightarrow & \quad \{\,\text{transitivity of } =\,\} \\
 & \forall(x : \mathcal{D}\ f\ .\ f\ x = \bot \equiv g\ x = \bot) \\
\equiv & \quad \{\,\text{spec } \mathcal{D}\,\} \\
 & \forall(x : \mathcal{D}\ f\ .\ 0 \equiv g\ x = \bot) \\
\equiv & \quad \{\,\text{def } \neg, \neq\,\} \\
 & \forall(x : \mathcal{D}\ f\ .\ g\ x \neq \bot) \\
\equiv & \quad \{\,\text{spec } \mathcal{D}\,\} \\
 & \forall(x : \mathcal{D}\ f\ .\ x \in \mathcal{D}\ g) \\
\equiv & \quad \{\,\text{spec } \subseteq\,\} \\
 & \mathcal{D}\ f \subseteq \mathcal{D}\ g
\end{array}
$$

If $f \sqsubseteq g$, we also call f a *restriction* of g and g a *(domain) extension* of f.

2.7 Conclusions

We have introduced the Funmath notation and some basic mathematical concepts and operators in the language. Many notations common in mathematics, such as set and quantification notation, are subsumed in Funmath as applications of suitably defined higher order operators to abstractions and tuples. Type operators have been defined to construct types for simple functions and functions whose result type depends on the given argument.

Chapter 3

Overloading and Polymorphism

3.1 Introduction

The objective of this chapter is to give semantics to overloading and implicit polymorphism in Funmath. Cardelli and Wegner [19] define overloading as follows:

> In *overloading*, the same variable name is used to denote different functions and the context is used to decide which function is denoted by a particular instance of the name.

By implicit polymorphism they denote the form of universal polymorphism where the type parameter which determines the type of the argument for each application of the function is left implicit. This is opposed to explicit polymorphism where polymorphic functions have explicit type parameters. Funmath uses the same terminology for these concepts.

Explicit polymorphism is incorporated in Funmath by treating types as ordinary values. Consequently, product types can be used to type explicitly polymorphic functions. Implicit polymorphism can be seen as an infinite form of overloading: the same variable name is used for a family of functions which are different because their domains are different, but structurally similar because all functions have the same body and the types have the same structure.

Incorporating overloading and implicit polymorphism in Funmath *without* extending the language itself is not evident because in Funmath a name always refers to *one* object. The solution is to merge the different functions, which we want to denote by the same name, into *one* function and bind that function to the name. For this purpose, we will introduce the *recursive function merge* operator. The same idea has also been used in the $\lambda\&$-calculus [21, 20] to provide semantics for the CLOS style of object oriented programming We also define function type operators suitable for typing polymorphic and overloaded functions.

3.2 Recursive function merge

We use the function merge operator $\&$ to merge the functional behaviour of a family of objects into one object. We define the operator by:

$$\begin{array}{l}\textbf{def } (\&\text{—}) : \mathit{Fam}\ \mathcal{U} \to \mathcal{U} \\ \textbf{with } \forall(\ F : \mathit{Fam}\ \mathcal{U}\ . \\ \qquad \&F = \\ \qquad [F] \dagger F \in \mathit{Fam}\ \mathcal{F} \setminus \iota\ \varepsilon\ ?\ (y : \bigcup(\mathcal{D} \circ F)\ .\ \&(x : \mathcal{D}\ F\ .\ F\ x\ y)))\end{array}$$

We also define a variadic function merge notation:

$$\textbf{def } (\&...) := \&$$

The definition of $\&$ is recursive. Safe choice is the base case of the definition; if the given family is nonempty and constant, the safe choice operator is applied. This is natural, because if all objects in the given family are the same, then they have the same functional behaviour, so the behaviour of the merged object equals the behaviour of any object in the given family. If the given family contains only functions, then we merge the functions by taking the union of the domains of the functions, and for each member of the union, recursively merging the values returned by the functions for that member. Note that merging these values may fail. If, for example, $f\ 0 = 0$ and $g\ 0 = 1$, then $(f\ \&\ g)\ 0 = (0\ \&\ 1) = \bot$. If for a family F of objects all (recursive) applications of $\&$ are defined, then F is called *recursively compatible*. Formally, we define recursive compatibility using the *definedness order*, which is given by:

$$\begin{array}{l}\textbf{def } (\text{—} \sqsubseteq_* \text{—}) : \mathcal{U} \times \mathcal{U} \to \mathbb{B} \\ \textbf{with } \forall(x : \mathcal{U}; y : \mathcal{U}\ .\ (x \sqsubseteq_* y) \equiv (x\ \&\ y) = y) \\ \textbf{par } (\text{—} \sqsubseteq_* \text{—}) \wedge (\text{—} \sqsubseteq_* \text{—}) \\ \textbf{par } (\text{—} = \text{—}) \sqsubseteq_* (\text{—} = \text{—})\end{array}$$

The undefined object $\bot$ is the least defined object and the empty function ε is the least defined function:

$$\forall(x : \mathcal{U}\ .\ \bot \sqsubseteq_* x) \wedge \forall(f : \mathcal{F}\ .\ \varepsilon \sqsubseteq_* f) \tag{3.1}$$

A family of objects is recursively compatible if each object is less defined than the merging of all objects:

$$\begin{array}{l}\textbf{def } (\copyright_*\text{—}) : \mathit{Fam}\ \mathcal{U} \to \mathbb{B} \\ \textbf{with } \forall(F : \mathit{Fam}\ \mathcal{U}\ .\ \copyright_* F \equiv \forall(x : \{F\}\ .\ x \sqsubseteq_* \&F))\end{array}$$

From the definition of $\&$ we can derive that the variadic version satisfies:

$$\forall(f, g : \mathcal{F}\ .\ (f\ \&\ g) = (x : \mathcal{D}\ f \cup \mathcal{D}\ g\ .\ f\ x\ \&\ g\ x))$$

Note that if one of the applications $f\ x$ and $g\ x$ is undefined, then the merging yields the other one because

$$\begin{array}{lll} a \,\&\, \bot & & \\ \quad \equiv & \{\text{def } \&\} & \&(a, \bot) \\ \quad \equiv & \{\text{function equality}\} & \&(0 \mapsto a) \\ \quad \equiv & \{\text{def } \&\} & [0 \mapsto a] \\ \quad \equiv & \{\text{def } \mapsto\} & [a^{\bullet\iota}\ 0] \\ \quad \equiv & \{\text{def } [\text{—}]\} & a \end{array}$$

This derivation also holds for $a = \bot$, because

$$\bot \,\&\, \bot = \&\varepsilon = [\varepsilon] = \bot$$

Consequently, the domain types in the definition of the function merge operator can be replaced by $\mathcal{U}$ without changing the meaning:

$$\forall(F : Fam\ \mathcal{F} \wedge F \neq \varepsilon\ .\ \&F = (y : \mathcal{U}\ .\ \&(x : \mathcal{U}\ .\ F\ x\ y)))$$
$$\forall(f, g : \mathcal{F}\ .\ (f \,\&\, g) = (x : \mathcal{U}\ .\ f\ x \,\&\, g\ x))$$

From the last property also follows that the definedness order $\sqsubseteq_*$ is recursive on functions:

$$\forall(f, g : \mathcal{F}\ .\ f \sqsubseteq_* g \equiv \forall(x : \mathcal{U}\ .\ f\ x \sqsubseteq_* g\ x)) \tag{3.2}$$

The precedence of & is given by:

par $(\text{—} \,\&\, \text{—}) = (\text{—} \,\&\, \text{—})$
par $(\text{—} \mapsto \text{—}) \,\&\, (\text{—} \mapsto \text{—})$
par $(\text{—}^{\bullet}\text{—}) \,\&\, (\text{—}^{\bullet}\text{—})$
par $(\text{—} \rceil \text{—}) \,\&\, (\text{—} \rceil \text{—})$
par $(\text{—} \circ \text{—}) \,\&\, (\text{—} \circ \text{—})$

3.3 Open, partial and extended function types

In this section we introduce some additional function type operators which we will later need to type merged functions. The first type operator is a version of the product operator, defined in Section 2.5, that does not have the domain restriction on the members:

def $(\bigotimes\text{—}) : Fam\ \mathcal{T} \to \mathcal{P}\ \mathcal{F}$
with $\forall(F : Fam\ \mathcal{T}; f : \mathcal{F}\ .\ f \in \bigotimes F \equiv \forall(x : \mathcal{D}\ F\ .\ f\ x \in F\ x))$

We call this operator the *open product* operator, because the domain type of the product does not limit the domains of the members of the product. We also introduce an open version of the function arrow:

def $(— \circ\!\!\rightarrow —) : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{P}\ \mathcal{F}$
with $\forall(A : \mathcal{T}; B : \mathcal{T}\ .\ (A \circ\!\!\rightarrow B) = \bigtimes\!\!\!\!\!\!\bigcirc(x : A\ .\ B))$

We can express the regular versions of the product and type arrow operators in terms of their open counterparts:

$$\forall(A : \mathcal{T}; B : \mathcal{T}\ .\ A \rightarrow B = (A \circ\!\!\rightarrow B) \cap ((\mathcal{U} \setminus A) \circ\!\!\rightarrow \iota\ \bot)) \tag{3.3}$$

$$\begin{aligned}\forall(F : Fam\ \mathcal{T}\ .\ \bigtimes F\ &= \bigtimes\!\!\!\!\!\!\bigcirc(x : \mathcal{U}\ .\ F\ x \dagger \iota\ \bot) \\ &= \bigtimes\!\!\!\!\!\!\bigcirc F \cap ((\mathcal{U} \setminus \mathcal{D}\ F) \circ\!\!\rightarrow \iota\ \bot))\end{aligned} \tag{3.4}$$

It's also possible to express open function types as a union of an infinite collection of regular function types having larger domain types:

$$\forall(A : \mathcal{T}; B : \mathcal{T}\ .\ A \circ\!\!\rightarrow B = \bigcup(C : \mathcal{T} \wedge A \subseteq C\ .\ C \rightarrow B))$$
$$\forall(F : Fam\ \mathcal{T}\ .\ \bigtimes\!\!\!\!\!\!\bigcirc F = \bigcup(G : Fam\ \mathcal{T} \wedge F \sqsubseteq G\ .\ \bigtimes G))$$

The open function type operators have some very nice subtyping properties:

$$\forall(F, G : (Fam\ \mathcal{T})^2\ .\ \mathcal{D}\ G \subseteq \mathcal{D}\ F \wedge \forall(x : \mathcal{D}\ G\ .\ F\ x \subseteq G\ x) \Rightarrow \bigtimes\!\!\!\!\!\!\bigcirc F \subseteq \bigtimes\!\!\!\!\!\!\bigcirc G) \tag{3.5}$$

$$\forall(A, A', B, B' : \mathcal{T}^4\ .\ A' \subseteq A \wedge B \subseteq B' \Rightarrow (A \circ\!\!\rightarrow B) \subseteq (A' \circ\!\!\rightarrow B')) \tag{3.6}$$

Note that in these subtyping rules, the order of the domain types is opposite to the order of the function types. Therefore, these subtyping rules are called *contravariant* in their domain types.

The superscript operator for types used above will be defined in Section 4.2, but until then, it suffices to know that

$$\forall(n : \mathbb{N}; A : \mathcal{T}\ .\ A^n = \{m : \mathbb{N} \mid m < n\} \rightarrow A)$$

In Section 2.5, we noted that the absence of $\bot$ in the codomain types of product types and arrow types, implies that the members of the function type are defined on the given domain type. Similar properties also hold for the open versions:

$$\forall(F : Fam\ \mathcal{T}; x : \mathcal{D}\ F; f : \bigtimes\!\!\!\!\!\!\bigcirc F\ .\ \bot \notin F\ x \Rightarrow x \in \mathcal{D}\ f)$$
$$\forall(A, B : \mathcal{T}^2; f : A \circ\!\!\rightarrow B\ .\ \bot \notin B \Rightarrow A \subseteq \mathcal{D}\ f)$$

The following type operators are convenient in practice, because they control the presence of $\bot$ in the codomain types, and as a consequence also control the domain information supplied by the type. For *partial function types*, we use the *partial product* and the *partial function arrow* operators, which are defined by:

def $(\bigtimes —) : Fam\ \mathcal{T} \rightarrow \mathcal{P}\ \mathcal{F}$
with $\forall(F : Fam\ \mathcal{T}\ .\ \bigtimes F = \bigtimes(x : \mathcal{D}\ F\ .\ F\ x \cup \iota\ \bot))$

def $(— \rightharpoonup —) : \mathcal{T}^2 \rightarrow \mathcal{P}\ \mathcal{F}$
with $\forall(A, B : \mathcal{T}^2\ .\ (A \rightharpoonup B) = A \rightarrow B \cup \iota\ \bot)$

The function types denoted by these operators contain functions which are *at most* defined on the given domain type. This is demonstrated by the following properties:

$$\forall(F : Fam\ \mathcal{T}; f : \mathcal{F}\ .\ f \in \bigtimes F \equiv \mathcal{D}\ f \subseteq \mathcal{D}\ F \wedge \forall(x : \mathcal{D}\ f\ .\ f\ x \in F\ x))$$
$$\forall(A, B : \mathcal{T}^2; f : \mathcal{F}\ .\ f \in (A \rightharpoonup B) \equiv \mathcal{D}\ f \subseteq A \wedge \{f\} \subseteq B)$$

These operators also have some nice subtyping properties:

$$\forall(F, G : (Fam\ \mathcal{T})^2\ .\ \mathcal{D}\ F \subseteq \mathcal{D}\ G \wedge \forall(x : \mathcal{D}\ F\ .\ F\ x \subseteq G\ x) \Rightarrow \bigtimes F \subseteq \bigtimes G) \tag{3.7}$$
$$\forall(A, A', B, B' : \mathcal{T}^4\ .\ A \subseteq A' \wedge B \subseteq B' \Rightarrow (A \rightharpoonup B) \subseteq (A' \rightharpoonup B')) \tag{3.8}$$

Note that in these subtyping rules, the domain types are in the same order as the function types. Hence, these rules are called *covariant*.

The following type operators are the opposite of the partial function type operators, and are therefore called *extended function type* operators. Instead of adding $\perp$ to the codomain type, they remove $\perp$:

def $(\bigtimes\!\!-) : Fam\ \mathcal{T} \to \mathcal{P}\ \mathcal{F}$
with $\forall(F : Fam\ \mathcal{T}\ .\ \bigtimes F = \bigotimes(x : \mathcal{D}\ F\ .\ F\ x \setminus \iota\ \perp))$

def $(-\!\!\rightharpoonup\!\!-) : \mathcal{T}^2 \to \mathcal{P}\ \mathcal{F}$
with $\forall(A, B : \mathcal{T}^2\ .\ (A \rightharpoonup B) = (A \circ\!\!\rightarrow (B \setminus \iota\ \perp)))$

This results in function types, whose functions are *at least* defined on the given domain type, which is demonstrated by:

$$\forall(F : Fam\ \mathcal{T}; f : \mathcal{F}\ .\ f \in \bigtimes F \equiv \mathcal{D}\ F \subseteq \mathcal{D}\ f \wedge \forall(x : \mathcal{D}\ F\ .\ f\ x \in F\ x))$$
$$\forall(A, B : \mathcal{T}^2; f : \mathcal{F}\ .\ f \in (A \rightharpoonup B) \equiv A \subseteq \mathcal{D}\ f \wedge \{f \rceil A\} \subseteq B)$$

The extended function type operators inherit the contravariant subtyping properties from the open function type operators.

All type arrow operators associate to the right and have lower precedence than Cartesian product:

par $- \to (- \circ\!\!\to (- \rightharpoonup (- \rightharpoonup (- \to -))))$
par $(- \times -) \circ\!\!\to (- \times -)$
par $(- \times -) \rightharpoonup (- \times -)$
par $(- \times -) \rightharpoonup (- \times -)$

3.4 Overloading

The combination of open function types and type intersection makes it possible to denote types for overloaded functions. Type systems combining function types and type intersection have been presented in [5, 3]. To overload the name f, so that on the type A the function f behaves like the function $f' : A \to B$, and on the type C the function f behaves like the function $f'' : C \to D$, we define f by:

$$\textbf{def } f : (A \circ\!\!\to B) \cap (C \circ\!\!\to D) \textbf{ with } f = f' \,\&\, f''$$

The type of f specifies that $\forall(x : A \,.\, f\ x \in B)$ and $\forall(x : C \,.\, f\ x \in D)$. From $f = f' \,\&\, f''$ follows that $\mathcal{D}\ f \subseteq A \cup C$. If $\bot \notin B$ and $\bot \notin D$ then we also have $A \cup C \subseteq \mathcal{D}\ f$. Together this yields $\mathcal{D}\ f = A \cup C$, so the definition only is correct if f' and f'' are recursively compatible. Recursive compatibility of the components is a natural requirement for overloading, because if the functional behaviour of the components is different in an identical context, then it is unsafe to use the same name for these components in that context.

Note that the types $(A \to B) \cap (C \to D)$ and $(A \to B) \cup (C \to D)$ cannot serve as type for f. If f were in the type $(A \to B) \cap (C \to D)$, then f would satisfy $\mathcal{D}\ f \subseteq A \cap C$, which is too restrictive. If f were in the type $(A \to B) \cup (C \to D)$, then we would have that $\mathcal{D}\ f \subseteq A$ or $\mathcal{D}\ f \subseteq C$, which also doesn't allow f to be defined on *both* A and C. The only valid regular arrow type for f is $A \cup C \to B \cup D$, but this type does not express the fact that the type of $f\ x$ depends on the type of x; for $x : A$ we can only derive that $f\ x \in B \cup D$, so we lose type information.

Using specifications, we can also define overloaded objects incrementally:

$$\textbf{spec } f : A \circ\!\!\to B \textbf{ with } f' \sqsubseteq_* f$$
$$\textbf{spec } f : C \circ\!\!\to D \textbf{ with } f'' \sqsubseteq_* f$$

From these specifications we cannot derive anymore that $\mathcal{D}\ f \subseteq A \cup C$, so they do not define f uniquely, but leave room for additional specifications for f on other types.

The manner of defining overloaded functions demonstrated above, also works for Curried functions, i.e. functions having more than one argument, because the function merge operator is recursive.

3.5 Implicit polymorphism

Implicit polymorphism can be seen as an infinite form of overloading, because the same name is used for an infinite collection of structurally similar functions. These functions only differ in type, which makes it possible to define the polymorphic version with type variables. As an illustration, we define the polymorphic function *twice*, which composes a given function with itself, by:

$$\textbf{def } \mathit{twice} : \bigcap(A : \mathcal{T} \,.\, (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A))$$
$$\textbf{with } \mathit{twice} = \&(A : \mathcal{T} \,.\, f : A \circ\!\!\to A \,.\, f \circ f)$$

Note that the family $A : \mathcal{T} \;.\; f : A \circ\!\!\to A \;.\; f \circ f$, which for every type A contains the *twice* function on $A \circ\!\!\to A$, is recursively compatible, because the innermost body $f \circ f$ does not depend on the type variable A.

By using the open arrow type $A \circ\!\!\to A$ as domain type instead of $A \to A$, we can make polymorphic type derivations; it is possible to make the following simple type derivation which shows that *twice twice* has the same type as *twice*:

$$
\begin{array}{ll}
 & \mathit{twice} \in \bigcap(A : \mathcal{T} \;.\; (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \\
\equiv & \qquad \{\,\text{def}\ \textstyle\bigcap\,\} \\
 & \forall(A : \mathcal{T} \;.\; \mathit{twice} \in (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \\
\equiv & \qquad \{\,\text{logic}\,\} \\
 & \forall(A : \mathcal{T} \;.\; \mathit{twice} \in ((A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \circ\!\!\to ((A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \wedge \\
 & \qquad\qquad \mathit{twice} \in (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \\
\Rightarrow & \qquad \{\,\text{def}\ \circ\!\!\to\,\} \\
 & \forall(A : \mathcal{T} \;.\; \mathit{twice}\ \mathit{twice} \in (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \\
\equiv & \qquad \{\,\text{def}\ \textstyle\bigcap\,\} \\
 & \mathit{twice}\ \mathit{twice} \in \bigcap(A : \mathcal{T} \;.\; (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A))
\end{array}
$$

This derivation would not have been possible if *twice* used the type $A \to A$ instead of $A \circ\!\!\to A$.

Using open arrow types as domain types instead of regular arrow types has two other consequences. The first consequence is that we lose some domain information during type checking: for $f : A \to A$ with $\bot \notin A$ we only can derive that $\mathit{twice}\ f \in A \circ\!\!\to A$, which does not imply $\mathcal{D}\ (\mathit{twice}\ f) = A$. For $x : A$ we can still derive that $\mathit{twice}\ f\ x$ is defined and has type A, but if $x \notin A$ then we cannot derive that $\mathit{twice}\ f\ x$ is undefined. We say that the type of *twice* only contains *positive* type information.

Sometimes, it is possible to derive some negative type information too. For instance, if the function f is strict, i.e. $f\ \bot = \bot$, then $f \in \iota\ \bot \circ\!\!\to \iota\ \bot$. From the type of *twice* it now follows that $\mathit{twice}\ f \in \iota\ \bot \circ\!\!\to \iota\ \bot$, so $\mathit{twice}\ f$ is strict too. This shows that in Funmath, strictness analysis is a special case of type inference.

The other consequence of using open arrow types is that the domain of *twice*, is much larger: we have that the domain of *twice* includes $\bigcup(A : \mathcal{T} \;.\; A \circ\!\!\to A)$, so that *twice* can be applied to *any* function of type $A \circ\!\!\to A$ for some A. This actually implies that *twice* can be applied to every function, because every function is in the type $\emptyset \circ\!\!\to \emptyset$. Therefore, the image definition of *twice* can just as well be given by $\mathit{twice} = (f : \mathcal{F} \;.\; f \circ f)$.

In the opposite direction, we can also give polymorphic definitions for general operators having universal types. We will illustrate this by giving a polymorphic definition for the function composition operator, which is defined in Section 2.6 by:

$$
\begin{array}{l}
\textbf{def}\ (\text{—} \circ \text{—}) : \mathcal{F} \times \mathcal{F} \to \mathcal{F} \\
\textbf{with}\ \forall(g : \mathcal{F}; f : \mathcal{F} \;.\; (g \circ f) = (x : \mathcal{D}\ f \;.\; g\ (f\ x)))
\end{array}
$$

Using open and extended arrow types and recursive function merge, we can give a following polymorphic definition, which enables us to infer types for applications of the function composition operator.

$$\textbf{def}\ (—\circ—) : \bigcap(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B) \circ\!\!\rightarrow (A \circ\!\!\rightarrow C))$$
$$\textbf{with}\ (\circ) = \&(A, B, C : \mathcal{T}^3\ .\ g : B \circ\!\!\rightarrow C; f : A \rightsquigarrow B\ .\ x : A\ .\ g\ (f\ x))$$

Note that the argument f has type $A \rightsquigarrow B$ and therefore the application $f\ x$ in the body is always defined. In the original definition this is achieved by the binding $x : \mathcal{D}\ f$. Note that using $A \circ\!\!\rightarrow B$ as type for f corresponds to using $\mathcal{U}$ as type for x in the original definition.

We show that the new definition indeed equals the original definition:

$$\begin{array}{ll}
 & \&(A, B, C : \mathcal{T}^3\ .\ g : B \circ\!\!\rightarrow C; f : A \rightsquigarrow B\ .\ x : A\ .\ g\ (f\ x)) \\
= & \quad \{\text{def } \&\} \\
 & g, f : \bigcup(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B))\ . \\
 & \&(A, B, C : \mathcal{T}^3 \wedge (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B)\ .\ x : A\ .\ g\ (f\ x)) \\
= & \quad \{\text{def } \&\} \\
 & g, f : \bigcup(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \circ\!\!\rightarrow B))\ . \\
 & x : \bigcup(A, B, C : \mathcal{T}^3 \wedge (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B)\ .\ A)\ . \\
 & \&(A, B, C : \mathcal{T}^3 \wedge (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B) \wedge x \in A\ .\ g\ (f\ x)) \\
= & \quad \{\text{def } \&\} \\
 & g, f : \bigcup(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \circ\!\!\rightarrow B))\ . \\
 & x : \bigcup(A, B, C : \mathcal{T}^3 \wedge (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B)\ .\ A)\ . \\
 & [A, B, C : \mathcal{T}^3 \wedge (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B) \wedge x \in A\ .\ g\ (f\ x)] \\
= & \quad \{\text{def } [—]\} \\
 & g, f : \bigcup(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B))\ . \\
 & x : \bigcup(A, B, C : \mathcal{T}^3 \wedge (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B)\ .\ A)\ .\ g\ (f\ x) \\
= & \quad \{\text{Lemma 1, Lemma 2}\} \\
 & g, f : \mathcal{F} \times \mathcal{F}\ .\ x : \mathcal{D}\ f\ .\ g\ (f\ x)
\end{array}$$

The typing lemmas 1 and 2 show that the union types for the arguments equal the types in the original abstraction:

Lemma 1

$$\bigcup(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B)) = \mathcal{F} \times \mathcal{F}$$

Proof

$$\begin{array}{lll}
(\supseteq)\ \mathcal{F} \times \mathcal{F} & & \\
\quad = & \{\text{property } \circ\!\!\rightarrow,\ \rightsquigarrow\} & (\emptyset \circ\!\!\rightarrow \emptyset) \times (\emptyset \rightsquigarrow \emptyset) \\
\quad \subseteq & \{\text{set theory}\} & \bigcup(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\rightarrow C) \times (A \rightsquigarrow B))
\end{array}$$

$$
\begin{array}{lll}
(\subseteq)\ (\circ\!\!\rightarrow) \in \mathcal{T}^2 \to \mathcal{P}\ \mathcal{F} \wedge (\rightharpoonup) \in \mathcal{T}^2 \to \mathcal{P}\ \mathcal{F} & & \\
\quad \Rightarrow & \{\text{def} \to\} & \forall(A, B, C : \mathcal{T}^3 \,.\, B \circ\!\!\rightarrow C \in \mathcal{P}\ \mathcal{F} \wedge A \rightharpoonup B \in \mathcal{P}\ \mathcal{F}) \\
\quad \equiv & \{\text{def } \mathcal{P}\} & \forall(A, B, C : \mathcal{T}^3 \,.\, B \circ\!\!\rightarrow C \subseteq \mathcal{F} \wedge A \rightharpoonup B \subseteq \mathcal{F}) \\
\quad \Rightarrow & \{\text{property } \times\} & \forall(A, B, C : \mathcal{T}^3 \,.\, (B \circ\!\!\rightarrow C) \times (A \rightharpoonup B) \subseteq \mathcal{F} \times \mathcal{F}) \\
\quad \equiv & \{\text{set theory}\} & \bigcup(A, B, C : \mathcal{T}^3 \,.\, (B \circ\!\!\rightarrow C) \times (A \rightharpoonup B)) \subseteq \mathcal{F} \times \mathcal{F}
\end{array}
$$

Lemma 2

$$\forall(f, g : \mathcal{F} \times \mathcal{F} \,.\, \bigcup(A, B, C : \mathcal{T}^3 \mathbin{\&} (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightharpoonup B) \,.\, A) = \mathcal{D}\ f)$$

Proof

$$
\begin{array}{ll}
 & x \in \bigcup(A, B, C : \mathcal{T}^3 \mathbin{\&} (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightharpoonup B) \,.\, A) \\
\equiv & \quad \{\text{def} \bigcup\} \\
 & \exists(A, B, C : \mathcal{T}^3 \mathbin{\&} (g, f) \in (B \circ\!\!\rightarrow C) \times (A \rightharpoonup B) \,.\, x \in A) \\
\equiv & \quad \{\text{def} \times\} \\
 & \exists(A, B, C : \mathcal{T}^3 \mathbin{\&} g \in B \circ\!\!\rightarrow C \wedge f \in A \rightharpoonup B \,.\, x \in A) \\
\equiv & \quad \{(\Leftarrow),(\Rightarrow)\} \\
 & x \in \mathcal{D}\ f
\end{array}
$$

The last step is proved in both directions in the following two sublemmas:

($\Rightarrow$) For $A, B, C : \mathcal{T}^3 \mathbin{\&} g \in B \circ\!\!\rightarrow C \wedge f \in A \rightharpoonup B$ we have:

$$
\begin{array}{lll}
x \in A & & \\
\quad \Rightarrow & \{f \in A \rightharpoonup B, \text{def} \rightharpoonup\} & f\ x \in B \wedge f\ x \neq \bot \\
\quad \Rightarrow & \{\text{logic}\} & f\ x \neq \bot \\
\quad \equiv & \{\text{spec } \mathcal{D}\} & x \in \mathcal{D}\ f
\end{array}
$$

So we have

$$
\begin{array}{ll}
 & \forall(A, B, C : \mathcal{T}^3 \mathbin{\&} g \in B \circ\!\!\rightarrow C \wedge f \in A \rightharpoonup B \,.\, x \in A \Rightarrow x \in \mathcal{D}\ f) \\
\equiv & \quad \{\text{logic}\} \\
 & \exists(A, B, C : \mathcal{T}^3 \mathbin{\&} g \in B \circ\!\!\rightarrow C \wedge f \in A \rightharpoonup B \,.\, x \in A) \Rightarrow x \in \mathcal{D}\ f
\end{array}
$$

($\Leftarrow$) Suppose $x \in \mathcal{D}\ f$
Then take $A := \iota\ x; B := \mathcal{U}; C := \mathcal{U}$ so that we have:

$$
\begin{array}{lll}
f \in A \rightharpoonup B & & \\
\quad \equiv & \{\text{def} \rightharpoonup\} & \forall(z : A \,.\, f\ z \in B \wedge f\ z \neq \bot) \\
\quad \equiv & \{A = \iota\ x,\ B = \mathcal{U}\} & \forall(z : \iota\ x \,.\, f\ z \in \mathcal{U} \wedge f\ z \neq \bot) \\
\quad \equiv & \{\text{one point rule } \forall\} & f\ x \in \mathcal{U} \wedge f\ x \neq \bot \\
\quad \equiv & \{x \in \mathcal{D}\ f\} & f\ x \in \mathcal{U} \\
\quad \equiv & \{\text{def } \mathcal{U}\} & 1
\end{array}
$$

$g \in B \circ\!\!\rightarrow C$

$\equiv$ {def $\circ\!\!\rightarrow$} $\quad \forall(z : B \;.\; g\ z \in C)$

$\equiv$ {$B = \mathcal{U},\ C = \mathcal{U}$} $\quad \forall(z : \mathcal{U} \;.\; g\ z \in \mathcal{U})$

$\equiv$ {def $\mathcal{U}$} $\quad \forall(z : \mathcal{U} \;.\; 1)$

$\equiv$ {logic} $\quad 1$

$x \in A$

$\equiv$ {$A = \iota\ x$} $\quad x \in \iota\ x$

$\equiv$ {def ι} $\quad x = x$

$\equiv$ {logic} $\quad 1$

So we have:
$x \in \mathcal{D}\ f \Rightarrow \exists(A, B, C : \mathcal{T}^3 \wedge g \in B \circ\!\!\rightarrow C \wedge f \in A \rightharpoonup B \;.\; x \in A)$

This completes the proof that both object definitions for $\circ$ are equivalent.

Extended arrow types can also be used as types for Curried polymorphic functions. The K combinator, for instance, is defined by:

def $K : \bigcap(A, B : \mathcal{T}^2 \;.\; A \circ\!\!\rightarrow B \circ\!\!\rightarrow A)$
with $K = \&(A, B : \mathcal{T}^2 \;.\; x : A \;.\; y : B \;.\; x)$

The type of this definition specifies that $\forall(A, B : \mathcal{T}^2; x : A; y : B \;.\; K\ x\ y \in A)$, and for the body we can show that $K = (x : \mathcal{U} \;.\; y : \mathcal{U} \;.\; x)$ by a similar derivation as we used in the previous example.

3.6 Explicit polymorphism

The following property shows that every open product type can also be written as an intersection of open arrow types with singleton domain types:

$$\forall(F : Fam\ \mathcal{T} \;.\; \bigtimes F = \bigcap(x : \mathcal{D}\ F \;.\; \iota\ x \circ\!\!\rightarrow F\ x)) \tag{3.9}$$

Proof of (3.9)

$f \in \bigcap(x : \mathcal{D}\ F \;.\; \iota\ x \circ\!\!\rightarrow F\ x)$

$\equiv$ {def $\bigcap$}

$\forall(x : \mathcal{D}\ F \;.\; f \in \iota\ x \circ\!\!\rightarrow F\ x)$

$\equiv$ {def $\circ\!\!\rightarrow$}

$\forall(x : \mathcal{D}\ F \;.\; \forall(y : \iota\ x \;.\; f\ y \in F\ x))$

$\equiv$ {one point rule $\forall$}

$$
\begin{array}{ll}
 & \forall(x : \mathcal{D}\ F\ .\ f\ x \in F\ x) \\
\equiv & \quad \{\, \text{def } \bigtimes \,\} \\
 & f \in \bigtimes F
\end{array}
$$

This property can be used to preserve type dependency in polymorphic type derivations. For example, if we compose the polymorphic function *twice* with the parametric identity function $id' : \bigtimes(A : \mathcal{T}\ .\ A \to A)$, defined in Section 2.5, then we can derive that the composition $twice \circ id'$ has an extended dependent type:

$$
\begin{array}{ll}
 & id' \in \bigtimes(A : \mathcal{T}\ .\ A \to A) \\
\Rightarrow & \quad \{\, (3.4) \,\} \\
 & id' \in \bigtimes(A : \mathcal{T}\ .\ A \to A) \\
\equiv & \quad \{\, (3.9) \,\} \\
 & id' \in \bigcap(A : \mathcal{T}\ .\ \iota\ A \circ\!\!\to A \to A) \\
\equiv & \quad \{\, \text{def } \bigcap \,\} \\
 & \forall(A : \mathcal{T}\ .\ id' \in \iota\ A \circ\!\!\to A \to A) \\
\Rightarrow & \quad \{\, (3.3),(3.6) \,\} \\
 & \forall(A : \mathcal{T}\ .\ id' \in \iota\ A \circ\!\!\to A \circ\!\!\to A) \\
\equiv & \quad \{\, \bot \notin A \circ\!\!\to A,\ \text{def } \rightharpoonup \,\} \\
 & \forall(A : \mathcal{T}\ .\ id' \in \iota\ A \rightharpoonup A \circ\!\!\to A) \\
\equiv & \quad \{\, twice \in \bigcap(A : \mathcal{T}\ .\ (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A)) \,\} \\
 & \forall(A : \mathcal{T}\ .\ twice \in (A \circ\!\!\to A) \circ\!\!\to (A \circ\!\!\to A) \wedge id' \in \iota\ A \rightharpoonup A \circ\!\!\to A) \\
\equiv & \quad \{\, (\circ) \in \bigcap(A, B, C : \mathcal{T}^3\ .\ (B \circ\!\!\to C) \times (A \rightharpoonup B) \circ\!\!\to (A \circ\!\!\to C)),\ \text{def } \circ\!\!\to \,\} \\
 & \forall(A : \mathcal{T}\ .\ twice \circ id' \in \iota\ A \circ\!\!\to A \circ\!\!\to A) \\
\equiv & \quad \{\, \text{def } \bigcap \,\} \\
 & twice \circ id' \in \bigcap(A : \mathcal{T}\ .\ \iota\ A \circ\!\!\to A \circ\!\!\to A) \\
\equiv & \quad \{\, (3.9) \,\} \\
 & twice \circ id' \in \bigtimes(A : \mathcal{T}\ .\ A \circ\!\!\to A)
\end{array}
$$

This derivation shows that applying $twice \circ id'$ to a type A, yields a function of type $A \circ\!\!\to A$. So we have preserved the type dependency of id' in the type derived for $twice \circ id'$. Property (3.9) also shows that adding the singleton set operator ι to a type system for implicit polymorphism, makes explicit polymorphic typing possible. However, checking whether an object has a singleton type, e.g. $x \in \iota\ y$, is the same as checking whether $x = y$, which is only possible to a limited extent if type checking has to be done statically.

3.7 Conclusions

We have presented a function merge operator and several function type operators, which can be used for defining polymorphic functions in Funmath. Especially the intersection of

open function types yields function types which are very convenient for expressing positive type information of functions. For overloaded functions, the number of different types is finite, which results in finite intersection types, which can be denoted by applications of the type intersection operator to tuples of types. Polymorphic functions have an infinite number of types which all have the same structure, which results in infinite intersections, which are denoted by applications of the type intersection operator to abstractions yielding types.

In this chapter, no algorithms for type checking and type inferation are given. The type system presented is so complex that automatic type checking and inferation is not possible. Therefore, a subset of the Funmath types needs to be determined for which these typing algorithms do exist.

Chapter 4

Some General Mathematical Concepts

4.1 Introduction

This chapter uses Funmath to define some basic mathematical concepts which are also very useful in computer science. Most of the operators defined in this chapter will be used very frequently in the following chapters. Similar definitions for the same concepts can also be found in [17, 35, 41, 43]. The presentation in this chapter is the first one that takes advantage of the type operators for polymorphism defined in Chapter 3.

4.2 Sequences

Sequences are structures which are indexed by (an initial part of) $\mathbb{N}$ and whose components all have the same type. Sequences can have infinite length. Therefore, it is convenient to have an explicit constant ∞, of which we only need to know that it is not a number:

spec $\infty : \mathcal{U}$ **with** $\infty \notin \mathbb{C}$

Now we extend the operators $<$ and $+$ so that they handle ∞ correctly:

spec $(<)$: $(\mathbb{R} \times \iota\infty \circ\!\!\to \iota\ 1) \cap$
$(\iota\infty \times \mathbb{R} \circ\!\!\to \iota\ 0) \cap$
$(\iota\infty \times \iota\infty \circ\!\!\to \iota\ 0)$

spec $(+) : \mathcal{F}$
with $\forall(x : \mathbb{R} \cup \iota\infty\ .\ x + \infty = \infty + x = \infty)$

To denote domains of sequences we now can introduce the operator $\square$ by

def $(\square\text{—}) : \mathbb{N} \cup \iota\infty \to \mathcal{P}\ \mathbb{N}$
with $\forall(n : \mathbb{N} \cup \iota\infty\ .\ \square n = \{m : \mathbb{N} \mid m < n\})$

Note that $\Box\infty = \mathbb{N}$. The type of sequences over A of length n can be defined using the following operator:

def $(\text{—}^{\frown}) : \mathcal{T} \times \mathbb{N} \cup \iota\infty \to \mathcal{T}$
with $\forall(A : \mathcal{T}; n : \mathbb{N} \cup \iota\infty \;.\; A^n = \Box n \to A)$

From a syntactic point of view, we regard $\text{—}^{\frown}$ as a postfix operator. Its ASCII pattern is `_ ^{ _ }`. Therefore, it has precedence over all infix operators.

Note that if $\perp$ is a member of A, the domains of the members of A^n are *subsets* of $\Box n$. For such A we therefore have that $A^n \subseteq A^{n+1}$. Only if $\perp$ is not in A, the domains of all members of A^n equal $\Box n$ which gives $A^n \cap A^{n+1} = \emptyset$.

For $n : \mathbb{N}$ we call A^n the type of *arrays* over A of length n. A^∞ is called the type of *streams* over A. A *list* over A is a sequence over A of finite length. A^* denotes the type of all lists over A:

def $(\text{—}^*) : \mathcal{T} \to \mathcal{T}$
with $\forall(A : \mathcal{T} \;.\; A^* = \bigcup(n : \mathbb{N} \;.\; A^n))$

Tuple notation can be used to denote arrays and lists. For instance, $(1, 2, 3) \in \mathbb{N}^3$ and all tuples are members of $\mathcal{U}^*$.

Finally, A^ω denotes the type of arbitrary sequences over A:

def $(\text{—}^\omega) : \mathcal{T} \to \mathcal{T}$
with $\forall(A : \mathcal{T} \;.\; A^\omega = A^* \cup A^\infty)$

We define the following operators on sequences. The *length* operator $\#$ returns the length of a sequence, which is ∞ in case the sequence is a stream:

def $(\#\text{—}) : \mathcal{U}^\omega \to \mathbb{N} \cup \iota\infty$
with $\forall(x : \mathcal{U}^\omega \;.\; \#x = min_{\mathbb{N}\cup\iota\infty,\leq}\ \{n : \mathbb{N} \mid x \in \mathcal{U}^n\})$

where $min_{\mathbb{N},\leq}$ is the minimum operator, which yields the smallest member of a set of natural numbers. The operator *min* will be introduced in Section 4.5.3.

The operator τ was already introduced before without definition. It can be used to create singleton lists. Here follows the definition:

def $(\tau\ \text{—}) : \bigcap(A : \mathcal{T} \;.\; A \circ\!\!\to A^1)$
with $\forall(a : \mathcal{U} \;.\; \tau\ a\ 0 = a)$

The *concatenation* operator $+\!\!+$ is used to append one sequence after another sequence:

spec $(\text{—} +\!\!+ \text{—}) : \bigcap(A : \mathcal{T}; m, n : (\mathbb{N} \cup \iota\infty)^2 \;.\; A^m \times A^n \circ\!\!\to A^{m+n})$
with $\forall(x : \mathcal{U}^\omega; y : \mathcal{U}^\omega; i : \Box(\#x + \#y) \;.\; (x +\!\!+ y)\ i = (i < \#x\ ?\ x\ i \dagger y\ (k - \#x)))$

Note that this specification defines ++ on sequences, but not for other objects. This makes it possible to overload ++ for other types, as described in Chapter 3. The operator α extracts the first element of a nonempty sequence, and the operator σ yields the rest of the sequence:

$$\textbf{spec } (\alpha\ \_), (\sigma\ \_) : \bigcap(A : \mathcal{T}; n : \mathbb{N} \cup \iota\infty\ .\ (A^{n+1} \circ\!\!\to A) \times (A^{n+1} \circ\!\!\to A^n))$$
$$\textbf{with } \forall(x : \mathcal{U}^\omega; i : \square(\#x)\ .\ \alpha\ x = x\ 0 \wedge \sigma\ x\ i = x\ (i+1))$$

The sequence *prefix* operator $\succ$ is now specified by:

$$\textbf{spec } (\_ \succ \_) : \bigcap(A : \mathcal{T}, n : \mathbb{N} \cup \iota\infty\ .\ A \times A^n \circ\!\!\to A^{n+1})$$
$$\textbf{with } \forall(x : \mathcal{U}^\omega\ .\ (\alpha\ x \succ \sigma\ x) = x)$$

The operator ω yields the last element of a nonempty list and the operator ρ everything but the last element of a list:

$$\textbf{spec } (\omega\ \_), (\rho\ \_) : \bigcap(A : \mathcal{T}; n : \mathbb{N}\ .\ (A^{n+1} \circ\!\!\to A) \times (A^{n+1} \circ\!\!\to A^n))$$
$$\textbf{with } \forall(x : \mathcal{U}^*; i : \square(\#x - 1)\ .\ \omega\ x = x\ (\#x - 1) \wedge \rho\ x\ i = x\ i)$$

The list *postfix* operator $\prec$ is given by:

$$\textbf{spec } (\_ \prec \_) : \bigcap(A : \mathcal{T}, n : \mathbb{N}\ .\ A^n \times A \circ\!\!\to A^{n+1})$$
$$\textbf{with } \forall(x : \mathcal{U}^*\ .\ (\rho\ x \prec \omega\ x) = x)$$

4.3 Direct extension of operators

An often used concept in Funmath is the extension of operators on values to operators on functions on those values. For instance, the direct extension of $+$ is denoted by $\hat{+}$ and satisfies $(f\ \hat{+}\ g)\ x = f\ x + g\ x$. Note that direct extension is not domain extension in the sense of Section 2.6. Direct extension doesn't enlarge the domain of the given operator, but it 'lifts' the domain to a function type over the original domain. In this section we give definitions for the direct extension operator $\hat{}$ and it's infix version $\_\ \hat{}\ \_$.

First we define the *function transposition* operator $\mathbf{T}$, which swaps the first and second argument of a nested function F so that $\mathbf{T}\ F\ y\ x = F\ x\ y$:

$$\textbf{def } \mathbf{T} : \bigcap(A, B, C : \mathcal{T}^3\ .\ (A \circ\!\!\to B \circ\!\!\to C) \circ\!\!\to (B \circ\!\!\to A \circ\!\!\to C))$$
$$\textbf{with } \mathbf{T} = (F : \mathcal{F}\ .\ y : \mathcal{U}\ .\ x : \mathcal{U}\ .\ F\ x\ y)$$

Note that the innermost abstraction in the definition of $\mathbf{T}$ is partial. Its domain is given by

$$x \in \mathcal{D}\ (\mathbf{T}\ F\ y) \equiv F\ x \in \mathcal{F} \wedge y \in \mathcal{D}\ (F\ x)$$

The function transposition operator is used to define the infix version of the *direct extension* operator, as follows:

$$\textbf{def } (— \hat{} —) : \bigcap(\ A, B, C, X : \mathcal{T}^3\ .\ (X \circ\!\!\to A) \times (A \times B \circ\!\!\to C) \times (X \circ\!\!\to B) \circ\!\!\to (X \circ\!\!\to C))$$
$$\textbf{with } (\hat{}) = (f : \mathcal{F}; op : \mathcal{F}; g : \mathcal{F}\ .\ op \circ \mathbf{T}\ (f, g))$$

This operator indeed shows the desired behaviour:

$$\begin{array}{lll} & (f\ \hat{+}\ g)\ y & \\ = & \{\text{def } \hat{}\ \} & ((+) \circ \mathbf{T}\ (f, g))\ y \\ = & \{\text{def } \circ\ \} & (+)\ (\mathbf{T}\ (f, g)\ y) \\ = & \{\text{def } \mathbf{T}\} & (+)\ (x : \mathcal{U}\ .\ (f, g)\ x\ y) \\ = & \{\text{function equality}\} & (+)\ (f\ y, g\ y) \\ = & \{\text{infix notation}\} & f\ y + g\ y \end{array}$$

The operator is given higher precedence than equality and associates to the left:

$$\textbf{par } (— \hat{} —) = (— \hat{} —)$$
$$\textbf{par } (— \hat{} —)\ \hat{} —$$

The infix version of the direct extension operator is generalized by replacing the pair of functions (f, g) by a nested function F:

$$\textbf{def } (\overset{<}{—}) : \bigcap(A, C, X, Y : \mathcal{T}^4\ .\ ((Y \circ\!\!\to A) \circ\!\!\to C) \circ\!\!\to (Y \circ\!\!\to X \circ\!\!\to A) \circ\!\!\to (X \circ\!\!\to C))$$
$$\textbf{with } (\overset{<}{—}) = (op : \mathcal{F}\ .\ F : \mathcal{F}\ .\ op \circ \mathbf{T}\ F)$$

This definition indeed generalizes the previous one, because $f\ \hat{op}\ g = \overset{<}{op}(f, g)$. The dyadic extension operator is still useful because of its better readable infix notation.

Because we defined the transposition operator $\mathbf{T}$ in the most general way, we can also apply direct extensions to families of functions with different domains. The domain of the application of $\overset{<}{op}$ to such a family depends on the strictness properties of op. For example, because the infix set union operator $\cup$ is strict in both arguments, i.e. $A \cup \bot = \bot \cup A = \bot$ for all $A : \mathcal{U}$, the direct extension yields functions whose domain is the intersection of the domains of the given functions:

$$\forall(f, g : (Fam\ \mathcal{T})^2\ .\ f\ \hat{\cup}\ g = (x : \mathcal{D}\ f \cap \mathcal{D}\ g\ .\ f\ x \cup g\ x))$$

On the other hand, the general non-strict union operator $\bigcup$, satisfying $\bigcup(A, \bot) = \bigcup(\bot, A) = A$ for $A : \mathcal{T}$ and $\bigcup(\bot, \bot) = \bigcup \varepsilon = \emptyset$, has a direct extension yielding universally defined functions:

$$\begin{array}{ll} \forall(f, g : (Fam\ \mathcal{T})^2\ .\ f\ \hat{\bigcup}\ g & = (x : \mathcal{U}\ .\ \bigcup(f\ x, g\ x)) \\ & = (x : \mathcal{U}\ .\ f\ x \cup g\ x \dagger f\ x \dagger g\ x \dagger \emptyset)) \end{array}$$

Another example of direct extension is the asymmetric function merge operator $\triangleright$, which is defined as the direct extension of the conditional operator $\dagger$ by:

$$\textbf{def } (— \triangleright —) : \mathcal{F}^2 \to \mathcal{F} \textbf{ with } (\triangleright) = \hat{\dagger}$$

This definition results in $(f \triangleright g) = (x : \mathcal{U} \;.\; f\ x \dagger g\ x)$, which shows that this function merge operator gives precedence to the first function. The strictness properties of $\dagger$ result in a direct extension which yields functions whose domain is the union of the given functions, i.e. $\mathcal{D}\ (f \triangleright g) = \mathcal{D}\ f \cup \mathcal{D}\ g$. We also define a function merge operator that gives precedence to the second function:

def $(— \triangleleft —) : \mathcal{F}^2 \to \mathcal{F}$ **with** $\forall(f, g : \mathcal{F}^2 \;.\; (f \triangleleft g) = (g \triangleright f))$

The syntactic precedence of these operators is given by:

par $(— \triangleleft —) \triangleleft —$
par $— \triangleright (— \triangleright —)$
par $(— \triangleright —) \mathbin{\hat{}} (— \triangleright —)$
par $(— \triangleleft —) \triangleright (— \triangleleft —)$
par $(— \,\&\, —) \triangleleft (— \,\&\, —)$

The recursive function merge operator $\&$, introduced in Section 3.2 can also be defined in terms of direct extensions:

$$\& = [] \triangleright (\overset{<}{\&}\,](\mathit{Fam}\ \mathcal{F} \setminus \iota\ \varepsilon))$$

This also is an illustration of how direct extension supports the variableless style of definition.

4.4 Predicates

In Funmath, predicates are functions with codomain $\mathbb{B}$. Therefore, the type $\mathit{Fam}\ \mathbb{B}$ is the type containing all predicates. Equivalent formulations for $\mathit{Fam}\ \mathbb{B}$ are $\bigcup(A : \mathcal{T} \;.\; A \to \mathbb{B})$ and $\mathcal{U} \rightharpoonup \mathbb{B}$. This section defines some useful operators on predicates.

The *unique existential quantifier* $\exists!$ yields true if there is exactly one domain value for which the given predicate is true:

def $(\exists!—) : \mathit{Fam}\ \mathbb{B} \to \mathbb{B}$
with $\forall(P : \mathit{Fam}\ \mathbb{B} \;.\; \exists!P \equiv \exists(x : \mathcal{D}\ P \;.\; P\ x \wedge \forall(y : \mathcal{D}\ P \wedge P\ y \;.\; x = y)))$

We also introduce the operator $(\!|—|\!)$ which yields a family satisfying a given predicate:

def $((\!|—|\!)) : \mathit{Fam}\ \mathbb{B} \to \mathit{Fam}\ \mathcal{U}$
with $\forall(P : \mathit{Fam}\ \mathbb{B} \;.\; (\!|\, P \,|\!) = (x : \mathcal{D}\ P \mid P\ x))$

The operator $\{\!|\ |\!\}$ yields the set of elements satisfying a predicate, and is defined by:

def $(\{\!|—|\!\}) : \mathit{Fam}\ \mathbb{B} \to \mathcal{T}$
with $\forall(P : \mathit{Fam}\ \mathbb{B}; x : \mathcal{U} \;.\; x \in \{\!|\, P \,|\!\} \equiv x \in \mathcal{D}\ P \wedge P\ x)$

The operator $[\![—]\!]$ yields the *unique* element satisfying a predicate, provided that there is such an element:

def $([\![—]\!]) : Fam\ \mathbb{B} \to \mathcal{U}$
with $[\![\ ]\!] = [\ ] \circ (\!|\ |\!)$

This operator satisfies $\mathcal{D}\ [\![\ ]\!] = \{P : Fam\ \mathbb{B} \mid \exists! P\} = \{\!|\exists!|\!\}$.

Sometimes, it is convenient to consider a partial predicate as a predicate which is false for the values for which it is not defined. For this purpose, we introduce the *predicate totalization* operator $\breve{—}$ which yields the domain extension of the given predicate that yields 0 for the values which are not in the domain of the given predicate:

def $(\breve{—}) : Fam\ \mathbb{B} \to \mathcal{U} \to \mathbb{B}$
with $\forall(P : \mathcal{U} \rightharpoonup \mathbb{B}\ .\ \breve{P} = P \triangleright 0^{\bullet}\mathcal{U})$

This operator satisfies:

$\forall(P : Fam\ \mathbb{B}; x : \mathcal{U}\ .\ \breve{P}\ x \equiv x \in \{\!|\ P\ |\!\})$

4.5 Relations and domain theory

Relations are a special case of predicates, which map pairs to truth values, which is in complete agreement with the functional notation used for relations; the expression $a < b$ is an application of the relation $— < —$ to the pair (a, b) returning a truth value. The operators on relations defined in this section, all are explicitly parametrized with the domain C of the relation. This is done because in Funmath many relations are extended to support variadic notation, or overloaded on different domains. Examples of such extended relations that are not only defined on pairs, are $=$ and $\neq$. Because these operators are not member of any relation type $C^2 \to \mathbb{B}$, the less restrictive type $C^2 \circ\!\!\to \mathbb{B}$ is used.

4.5.1 Relation notation

Some basic operators on relations can be obtained from corresponding operators on $\mathbb{B}$ using the direct extension operator $— \hat{—} —$ defined in Section 4.3. We will use direct extension in combination with the *infix notation* operator $—\langle—\rangle—$ which we define by

def $(—\langle—\rangle—) : \mathcal{U} \times \mathcal{F} \times \mathcal{U} \to \mathcal{U}$
with $\forall(f : \mathcal{F}; x, y : \mathcal{U}^2\ .\ (x \langle f \rangle y) = f\ (x, y))$

and give precedence over equality by

par $(—\langle—\rangle—) = (—\langle—\rangle—)$
par $(—\langle—\rangle—)\langle—\rangle—$

Using this operator we can turn any function into an infix operator by surrounding it with $\langle\ \rangle$. The combination of direct extension with the infix notation operator makes it possible to write:

$$x\langle R\ \hat{\vee}\ S\rangle y \equiv x\langle R\rangle y \vee x\langle S\rangle y$$
$$x\langle R\ \hat{\wedge}\ S\rangle y \equiv x\langle R\rangle y \wedge x\langle S\rangle y$$
$$x\langle R\ \hat{\Rightarrow}\ S\rangle y \equiv x\langle R\rangle y \Rightarrow x\langle S\rangle y$$

4.5.2 Properties of relations

We define the usual predicates for reflexivity, transitivity and anti-symmetry of a relation R on a given domain C:

def $reflexive : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow \mathbb{B})$
with $\forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .$
$\quad reflexive\ C\ R \equiv \forall(x : C\ .\ x\langle R\rangle x))$

def $transitive : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow \mathbb{B})$
with $\forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .$
$\quad transitive\ C\ R \equiv \forall(x, y, z : C^3\ .\ x\langle R\rangle y \wedge y\langle R\rangle z \Rightarrow x\langle R\rangle z))$

def $antisymmetric : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow \mathbb{B})$
with $\forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .$
$\quad antisymmetric\ C\ R \equiv \forall(x, y : C^2\ .\ x\langle R\rangle y \wedge y\langle R\rangle x \Rightarrow x = y))$

A relation that satisfies the three previous predicates on a given domain is called a *partial order* on that domain:

def $partial\_order : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow \mathbb{B})$
with $partial\_order = reflexive\ \hat{\hat{\wedge}}\ transitive\ \hat{\hat{\wedge}}\ antisymmetric$

Finally, we define the notion of *totality* of a relation by

def $total : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow \mathbb{B})$
with $\forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .$
$\quad total\ C\ R \equiv \forall(x, y : C^2\ .\ x\langle R\rangle y \vee y\langle R\rangle x))$

4.5.3 Minimal and maximal elements

We define the notion of *minimal element*, so that the value x is a minimal element of the set S with respect to the relation R, if x is contained in F and there is no element in F that is below x:

$$\textbf{def } is\_min : \bigtimes(C : \mathcal{T} \;.\; (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with } \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; S : \mathcal{P}\ C; x : C \;.$$
$$x\langle is\_min\ C\ R\rangle S \equiv x \in S \wedge \forall(y : S \;.\; \neg(y\langle R\rangle x)))$$

Note that $\{\!|\langle is\_min\ C\ R\rangle S|\!\}$ denotes the set of minimal elements of S with respect to R. If there is a unique minimal element, i.e. $\exists!(\langle is\_min\ C\ R\rangle S)$, then this element is denoted by $[\![\langle is\_min\ C\ R\rangle S]\!]$. The operator $min_{C,R}$ which yields this unique minimum can thus be defined by:

$$\textbf{spec } (min\_) : \bigcap(C : \mathcal{T} \;.\; \iota\ C \times (C^2 \circ\!\!\rightarrow \mathbb{B}) \circ\!\!\rightarrow (\mathcal{P}\ C \cup Fam\ C) \circ\!\!\rightarrow C \cup \iota\ \bot)$$
$$\textbf{with } \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; S : \mathcal{P}\ C; F : Fam\ C \;.$$
$$min_{C,R}\ S = [\![\langle is\_min\ C\ R\rangle S]\!] \wedge$$
$$min_{C,R}\ F = min_{C,R}\ \{F\})$$

Note that we also defined the minimum operator on families of objects, in which case it yields the minimum element in the range of the family.

To get similar operators for maximal elements, we introduce the *swap* operator, which reverses the argument pair of a given dyadic operator:

$$\textbf{def } (\overleftrightarrow{\_}) : \bigcap(A, B, C : \mathcal{T}^3 \;.\; (A \times B \circ\!\!\rightarrow C) \circ\!\!\rightarrow B \times A \circ\!\!\rightarrow C)$$
$$\textbf{with } \forall(op : \mathcal{F} \;.\; \overleftrightarrow{op} = (x, y : \mathcal{U}^2 \;.\; y\langle op\rangle x))$$

Now we can define the operators for maximal elements by first reversing the given relation and then applying the corresponding operator for minimal elements:

$$\textbf{def } is\_max : \bigtimes(C : \mathcal{T} \;.\; (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with } \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B} \;.\; is\_max\ C\ R = is\_min\ C\ \overleftrightarrow{R})$$
$$\textbf{spec } (max\_) : \bigcap(C : \mathcal{T} \;.\; \iota\ C \times (C^2 \circ\!\!\rightarrow \mathbb{B}) \circ\!\!\rightarrow (\mathcal{P}\ C \cup Fam\ C) \circ\!\!\rightarrow C \cup \iota\ \bot)$$
$$\textbf{with } \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B} \;.\; max_{C,R} = min_{C,\overleftrightarrow{R}})$$

4.5.4 Upper and lower bounds

In the same style as we introduced the notions of minimal and maximal elements, we also introduce *least upper bounds* and *greatest lower bounds*. An element x is an *upper bound* of a set S with respect to the relation R, if it is above all elements in S:

$$\textbf{def } is\_ub : \bigtimes(C : \mathcal{T} \;.\; (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with } \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; S : \mathcal{P}\ C; x : C \;.$$
$$x\langle is\_ub\ C\ R\rangle S \equiv \forall(y : S \;.\; y\langle R\rangle x))$$

An element x is a *greatest element* of a set S with respect to the relation R, if it is an upper bound of S which is contained in S:

$$\textbf{def}\ is\_greatest : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with}\ \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; S : \mathcal{P}\ C; x : C\ .$$
$$x\langle is\_greatest\ C\ R\rangle S \equiv x \in S \wedge x\langle is\_ub\ R\rangle S)$$

The notions for *lower bound* and *least element* are obtained by swapping the arguments of the relation:

$$\textbf{def}\ is\_lb : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with}\ \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .\ is\_lb\ C\ R = is\_ub\ C\ \overleftrightarrow{R})$$
$$\textbf{def}\ is\_least : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with}\ \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .\ is\_least\ C\ R = is\_greatest\ C\ \overleftrightarrow{R})$$

An element x is called the *least upper bound* of the set S with respect to the relation R, if it is the least element of the set of upper bound of S:

$$\textbf{def}\ is\_lub : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with}\ \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; S : \mathcal{P}\ C\ .$$
$$\langle is\_lub\ C\ R\rangle S = \langle is\_least\ C\ R\rangle\{\!|\ \langle is\_ub\ C\ R\rangle S\ |\!\})$$

The operator $\bigsqcup\_\!\!\_$ returns the unique least upper bound of a given set, if it exists:

$$\textbf{spec}\ (\bigsqcup\_\!\!\_) : \bigcap(C : \mathcal{T}\ .\ \iota\ C \times (C^2 \circ\!\!\rightarrow \mathbb{B}) \circ\!\!\rightarrow (\mathcal{P}\ C \cup Fam\ C) \circ\!\!\rightarrow C \cup \iota\ \bot)$$
$$\textbf{with}\ \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; S : \mathcal{P}\ C; Fam\ C\ .$$
$$\bigsqcup_{C,R}\ S = [\![\langle is\_lub\ C\ R\rangle S]\!] \wedge$$
$$\bigsqcup_{C,R}\ F = \bigsqcup_{C,R}\ \{F\})$$

The *bottom element* of a relation, is the element that is below all other elements in the relation. If a relation R has a unique bottom element, which we denote by $bot\ C\ R$, then this element is the least upper bound of the empty set:

$$\textbf{def}\ bot : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightharpoonup C)$$
$$\textbf{with}\ \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .\ bot\ C\ R = \bigsqcup_{C,R}\ \emptyset)$$

Using the swap operator we define the corresponding operators for greatest lower bounds and bottom elements:

$$\textbf{def}\ is\_glb : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times \mathcal{P}\ C \rightarrow \mathbb{B})$$
$$\textbf{with}\ \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .\ is\_glb\ C\ R = is\_lub\ C\ \overleftrightarrow{R})$$
$$\textbf{spec}\ (\bigsqcap\_\!\!\_) : \bigcap(C : \mathcal{T}\ .\ \iota\ C \times (C^2 \circ\!\!\rightarrow \mathbb{B}) \circ\!\!\rightarrow (\mathcal{P}\ C \cup Fam\ C) \circ\!\!\rightarrow C \cup \iota\ \bot)$$
$$\textbf{with}\ \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .\ \bigsqcap_{C,R} = \bigsqcup_{C,\overleftrightarrow{R}})$$
$$\textbf{def}\ (top\ \_\!\!\_) : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightharpoonup C)$$
$$\textbf{with}\ \forall(C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}\ .\ top\ C\ R = bot\ C\ \overleftrightarrow{R})$$

4.5.5 Least fixed points

An element x is a *fixed point* of a function f, if it satisfies $f\ x = x$. It is called a *least fixed point* if it is the least of all fixed points with respect to the relation R:

$$\begin{array}{l}\textbf{def}\ \mathit{is\_lfp} : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow \mathbb{B}) \rightarrow C \times (C \circ\!\!\rightarrow C) \rightarrow \mathbb{B}) \\ \textbf{with}\ \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; f : C \circ\!\!\rightarrow C\ . \\ \qquad \langle \mathit{is\_lfp}\ C\ R\rangle f = \langle \mathit{is\_least}\ C\ R\rangle \{\!|\, y : C\ .\ f\ y = y \,|\!\}\ )\end{array}$$

The operator *fix*— yields the least fixed point of a function, provided that it exists and is unique:

$$\begin{array}{l}\textbf{spec}\ (\mathit{fix}\text{—}) : \bigcap(C : \mathcal{T}\ .\ \iota\ C \times (C^2 \circ\!\!\rightarrow \mathbb{B}) \circ\!\!\rightarrow (C \circ\!\!\rightarrow C) \circ\!\!\rightarrow C \cup \iota\ \bot) \\ \textbf{with}\ \forall(\ C : \mathcal{T}; R : C^2 \circ\!\!\rightarrow \mathbb{B}; f : C \circ\!\!\rightarrow C\ . \\ \qquad \mathit{fix}_{C,R}\ f = [\![\langle \mathit{is\_lfp}\ C\ R\rangle f]\!]\ )\end{array}$$

4.6 Some notions from algebra

This section introduces some basic notions from algebra in the same style as the previous section. That is, the predicates and operators are parametrized with a domain type C over which the given operator is defined.

4.6.1 Monoids

An operator is called a *monoid* over a given domain if it is *associative* and has a *unit element*. Associativity is defined by:

$$\begin{array}{l}\textbf{def}\ \mathit{associative} : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow C) \rightarrow \mathbb{B}) \\ \textbf{with}\ \forall(\ C : \mathcal{T}; op : C^2 \circ\!\!\rightarrow C\ . \\ \qquad \mathit{associative}\ C\ op \equiv \forall(x, y, z : C^3\ .\ (a\langle op\rangle b)\langle op\rangle c = a\langle op\rangle(b\langle op\rangle c))\end{array}$$

The unit element predicate is given by:

$$\begin{array}{l}\textbf{def}\ \mathit{is\_unit} : \bigtimes(C : \mathcal{T}\ .\ C \times (C^2 \circ\!\!\rightarrow C) \rightarrow \mathbb{B}) \\ \textbf{with}\ \forall(\ C : \mathcal{T}; op : C^2 \circ\!\!\rightarrow C; e : C\ . \\ \qquad e\langle \mathit{is\_unit}\ C\rangle op \equiv \forall(a : C\ .\ a\langle op\rangle e = a \wedge e\langle op\rangle a = a))\end{array}$$

If an operator has a unit element on a domain C, then this element is denoted by $\mathit{unit}\ C\ op$:

$$\begin{array}{l}\textbf{def}\ \mathit{unit} : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\rightarrow C) \rightharpoonup C) \\ \textbf{with}\ \forall(\ C : \mathcal{T}; op : C^2 \circ\!\!\rightarrow C\ . \\ \qquad \mathit{unit}\ C\ op = [\![\langle \mathit{is\_unit}\ C\rangle op]\!]\ )\end{array}$$

The predicate *monoid* is now defined by:

$$\textbf{def}\ \mathit{monoid} : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\to C) \to \mathbb{B})$$
$$\textbf{with}\ \forall(\ C : \mathcal{T}; \mathit{op} : C^2 \circ\!\!\to C\ .$$
$$\mathit{monoid}\ C\ \mathit{op} \equiv \mathit{associative}\ C\ \mathit{op} \wedge \exists(\langle \mathit{is\_unit}\ C \rangle \mathit{op}))$$

For monoids we can define a *fold* operator. This operator maps a monoid to an operator on lists, which reduces a list of domain elements to a single domain element, by repeatedly applying the monoid:

$$\textbf{def}\ \mathit{fold} : \bigtimes(C : \mathcal{T}\ .\ \{\!|\ \mathit{monoid}\ C\ |\!\!\} \times C^* \to C)$$
$$\textbf{with}\ \forall(\ C : \mathcal{T}; \mathit{op} : \{\!|\ \mathit{monoid}\ C\ |\!\!\}\ .$$
$$\mathit{op}\langle \mathit{fold}\ C \rangle \varepsilon = \mathit{unit}\ C\ \mathit{op} \wedge$$
$$\forall(a : C\ .\ \mathit{op}\langle \mathit{fold}\ C \rangle \tau\ a = a) \wedge$$
$$\forall(x, y : (C^*)^2\ .\ \mathit{op}\langle \mathit{fold}\ C \rangle (x +\!\!+ y) = (\mathit{op}\langle \mathit{fold}\ C \rangle x)\langle \mathit{op} \rangle (\mathit{op}\langle \mathit{fold}\ C \rangle y)))$$

This operator is also called the *reduce* operator, and the application $\mathit{op}\langle \mathit{fold}\ S \rangle$ is called a *reduction* of *op*. A reduction that often occurs in practice is the reduction of the sequence concatenation operator $+\!\!+$. Therefore, we define the following shorthand *cat* for it:

$$\textbf{def}\ \mathit{cat} := (+\!\!+)\langle \mathit{fold}\ \mathcal{U}^\omega \rangle$$

The notion reduction does not only make sense for monoids, but can also be used for operators which are not associative or do not have a unit element. Because the operator need not be associative, we have to specify the reduction order, and because it need not have a unit element, we have to provide an alternative base case. This results in the following two operators for *directed reductions*.

$$\textbf{spec}\ (- \not\to_- -) : \bigcap(C, D : \mathcal{T}^2\ .\ (C \times D \circ\!\!\to C) \times C \times D^* \circ\!\!\to C)$$
$$\textbf{with}\ \forall(\ C, D : \mathcal{T}^2; \mathit{op} : C \times D \circ\!\!\to C; e : C\ .$$
$$(\mathit{op} \not\to_e \varepsilon) = e \wedge$$
$$\forall(a : D; x : D^*\ .\ (\mathit{op} \not\to_e (x \prec a)) = (\mathit{op} \not\to_e x)\langle \mathit{op} \rangle a))$$

This operator is the left to right fold operator, because it applies the operator from left to right, e.g.

$$(\mathit{op} \not\to_e (a, b, c)) = ((e\langle \mathit{op} \rangle a)\langle \mathit{op} \rangle b)\langle \mathit{op} \rangle c$$

The right to left fold operator applies the operator in the other direction:

$$\textbf{spec}\ (- \not\leftarrow_- -) : \bigcap(C, D : \mathcal{T}^2\ .\ (D \times C \circ\!\!\to C) \times C \times D^* \circ\!\!\to C)$$
$$\textbf{with}\ \forall(\ C, D : \mathcal{T}^2; \mathit{op} : C \times D \circ\!\!\to C; e : C\ .$$
$$(\mathit{op} \not\leftarrow_e \varepsilon) = e \wedge$$
$$\forall(a : D; x : D^*\ .\ (\mathit{op} \not\leftarrow_e (a \succ x)) = a\langle \mathit{op} \rangle (\mathit{op} \not\leftarrow_e x)))$$

This results in:

$$(op \not\leftarrow_e (a, b, c)) = a\langle op\rangle(b\langle op\rangle(c\langle op\rangle e))$$

We give the fold operators precedence over equality by

par $(- \not\rightarrow - -) = (- \not\rightarrow - -)$
par $(- \not\leftarrow - -) = (- \not\leftarrow - -)$

Note that because monoids are associative, we have that their left to right and right to left reductions are identical:

$$\forall(\ C : \mathcal{T}; op : \{\!|\ monoid\ C\ |\!\}\ .\ op\langle fold\ C\rangle = (op \rceil C^2) \not\rightarrow_{unit\ C\ op} = (op \rceil C^2) \not\leftarrow_{unit\ C\ op})$$

4.6.2 Groups

The notion *inverse element* is defined by

def $is\_inverse : \bigtimes(C : \mathcal{T}\ .\ \{\!|\ monoid\ C\ |\!\} \to C^2 \to \mathbb{B})$
with $\forall(\ C : \mathcal{T}; op : \{\!|\ monoid\ C\ |\!\}; x, y : C^2\ .$
$x\langle is\_inverse\ C\ op\rangle y \equiv x\langle op\rangle y = unit\ C\ op \wedge y\langle op\rangle x = unit\ C\ op)$

A *group* is a monoid in which every element has an inverse:

def $group : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\to C) \to \mathbb{B})$
with $\forall(\ C : \mathcal{T}; op : C^2 \circ\!\!\to C\ .$
$group\ C\ op \equiv monoid\ C\ op \wedge \forall(x : C\ .\ \exists(\langle is\_inverse\ C\ op\rangle x)))$

It is an easy exercise to show that every element of a group has a unique inverse. Therefore, the following *inverse* operator is well defined:

def $inverse : \bigtimes(C : \mathcal{T}\ .\ \{\!|\ group\ C\ |\!\} \to C \to C)$
with $\forall(\ C : \mathcal{T}; op : \{\!|\ group\ C\ |\!\}; x : C\ .$
$inverse\ C\ op\ x = [\![\langle is\_inverse\ C\ op\rangle x]\!])$

We define commutativity by

def $commutative : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\to C) \to \mathbb{B})$
with $\forall(\ C : \mathcal{T}; op : C^2 \circ\!\!\to C\ .$
$commutative\ C\ op \equiv op \rceil C^2 = \overleftrightarrow{op} \rceil C^2)$

A commutative group is called *Abelian*:

def $Abelian : \bigtimes(C : \mathcal{T}\ .\ (C^2 \circ\!\!\to C) \to \mathbb{B})$
with $Abelian = group \mathbin{\hat{\hat{\wedge}}} commutative$

4.7 Pattern matching

Pattern matching is a popular style of defining functions in functional programming languages, like Miranda. Functions are defined by giving a series of equation of the form $f\ (pattern) = expression$. The pattern is composed of variables, predefined constants and so called *constructors*, which are a special kind of constants used to construct members of user defined algebraic data types.

In [12] it has been shown how pattern matching can be done in Funmath using the conditional operators ? and ↾ by considering constructors as a special case of injective functions, whose inverse can be used to access the arguments of the constructor. This section presents a different way to define functions in Funmath by pattern matching, which uses the function merge operator & and the one-point function definer $\mapsto$ to construct the alternatives of the function, which then are joined with the asymmetric function merge operator $\triangleright$. For instance, if we have the data type X, with a constant $c : X$ and a constructor $C : X \to X \to X$, then the Funmath equivalent for the Miranda styled function definition

$$f\ c = e;$$
$$f\ (C\ x\ y) = E_{x,y} provided\ P_{x,y}$$

is

$$f = \ c \mapsto e \triangleright$$
$$\&(x, y : X^2 \mid P_{x,y}\ .\ C\ x\ y \mapsto E_{x,y})$$

This solution is slightly more general than the solution presented in [12], because injectivity of the constructor functions is not required. For instance, the second alternative in the definition of f is defined for all elements constructed by C, if

$$\forall(x, y, a, b : X^4 \mid P_{x,y} \wedge P_{a,b}\ .\ C\ x\ y = C\ a\ b \Rightarrow E_{x,y} = E_{a,b})$$

which is less restrictive than the injectivity requirement

$$\forall(x, y, a, b : X^4 \mid P_{x,y} \wedge P_{a,b}\ .\ C\ x\ y = C\ a\ b \Rightarrow x = a \wedge y = b)$$

Finally, note that because we use the asymmetric function merge operator $\triangleright$ to join the function alternatives, for a given argument the first matching alternative is chosen. If the domains of the alternatives are disjoint, then it is also safe to use the function merge operator & instead of $\triangleright$.

4.8 Conclusions

We have shown that a wide range of general mathematical concepts can be expressed in the Funmath language. General operators such as the least upper bound and least fixed point operator, have been defined as polymorphic higher order functions using the type

operators introduced in Chapter 2 and 3. The general operators have an explicit type parameter C which serves to specify in which domain the operator must be applied. In this way, we can handle relations and functions that are overloaded on different domains and which have different properties depending on the domain C that is chosen.

Chapter 5

Describing Grammars in Funmath

5.1 Introduction

This chapter will show how Funmath can be used to specify the syntax of a language. This method will be used in the next chapter to formally describe the syntax of Funmath itself. A language is represented as a set of sentences, where a sentence is a list of symbols from a chosen alphabet. Basic operators for the construction of complicated languages are defined as functions over sets.

5.2 Operators on languages

5.2.1 Types for languages

A *sentence* over an alphabet S is a list of symbols from S. Hence, the type of all sentences over S is S^*. Note that this implies that sentences have finite length. A *language* over S is a set of sentences over S. Thus, a language over S is a subset of S^*. So the type of all languages over S, which we will denote by $\mathcal{L}\ S$, is the set of all subsets of S^*:

def $\mathcal{L} : \mathcal{T} \to \mathcal{T}$
with $\forall(S : \mathcal{T}\ .\ \mathcal{L}\ S = \mathcal{P}\ (S^*))$

Note that $\forall(A : \mathcal{T}; B : \mathcal{T}\ .\ A \subseteq B \Rightarrow \mathcal{L}\ A \subseteq \mathcal{L}\ B)$ so $\mathcal{L}\ \mathcal{U}$ is the type containing all languages. This implies that $\mathcal{L}\ \mathcal{U} = \bigcup(S : \mathcal{T}\ .\ \mathcal{L}\ S) = \bigcup \mathcal{L}$.

5.2.2 Basic grammar operations

Because languages are sets, uniting languages can be done using the set union operator $\bigcup$; for all $A : \mathcal{L}\ S$ and $B : \mathcal{L}\ S$ we have that $A \cup B$ is the language containing all sentences from A and B, which therefore also has type $\mathcal{L}\ S$. The use of operator $\cup$ on languages corresponds to the use of the *alternative* operator | in BNF notation.

For concatenation of languages we define the operator *Cat*:

spec $Cat : \bigcap(S : \mathcal{T} \;.\; (\mathcal{L}\ S)^* \circ\!\!\rightarrow \mathcal{L}\ S)$
with $\forall(F : (\mathcal{L}\ \mathcal{U})^* \;.\; Cat\ F = \{cat \rceil \bigtimes F\})$

$Cat\ F$ contains the sentences constructed by taking a sentence from every language in F and then concatenating those sentences. So for $s : A$ and $t : B$ we have $s +\!\!+ t \in Cat\ (A, B)$.

In BNF notation, concatenation of a finite number of languages is done by juxtaposition. This is not possible in Funmath because juxtaposition is already used for default function application. For this purpose we introduce the variadic operator $\diamond$, which has precedence over $\cup$:

def $(\diamond ...) := Cat$
par $(— \diamond —) \cup (— \diamond —)$

Note that the unit element of language concatenation is $\iota\ \varepsilon$, the language containing the empty sentence only. The zero element of $\diamond$ is the empty set $\emptyset$, which also is the unit element of $\cup$. We introduce the names *Empty* and *Fail* for these constants respectively:

def $Empty := \iota\ \varepsilon$
def $Fail := \emptyset$

Note that for any alphabet S we have $Fail \in \mathcal{L}\ S$ because $\emptyset \subseteq S^*$, and also that $Empty \in \mathcal{L}\ S$ because $\varepsilon \in S^*$. These are the *only* languages having this property, i.e. we have

$$\bigcap \mathcal{L} = \bigcap(S : \mathcal{T} \;.\; \mathcal{L}\ S) = \mathcal{L}\ \emptyset = \{Empty, Fail\}$$

Terminal symbols are denoted by underlining, so that $\underline{a}$ is the language containing the sentence which only contains the symbol a:

def $(\underline{\quad}) : \bigcap(S : \mathcal{T} \;.\; S \circ\!\!\rightarrow \mathcal{L}\ S)$
with $\forall(a : \mathcal{U} \;.\; \underline{a} = \iota\ (\tau\ a))$

With the basic grammar operators for union, concatenation and terminal symbols, we have incorporated all BNF constructs in Funmath. In our approach a *nonterminal* is just a name bound to a language. Production rules are represented as equations whose left hand side is a nonterminal and whose right hand side is constructed from terminals and nonterminals using the operators $\diamond$ and $\cup$. Different production rules for the same nonterminal are joined together by defining the nonterminal as the union of the different productions. In this way, we can describe all *context-free* languages.

The following example shows how the language of natural numbers can be described using only the operators defined before. First we define the language *Digit* containing all decimal digits:

def $Digit : \mathcal{L}\ ASCII$
with $Digit = \underline{\texttt{'0'}} \cup \underline{\texttt{'1'}} \cup \underline{\texttt{'2'}} \cup \underline{\texttt{'3'}} \cup \underline{\texttt{'4'}} \cup \underline{\texttt{'5'}} \cup \underline{\texttt{'6'}} \cup \underline{\texttt{'7'}} \cup \underline{\texttt{'8'}} \cup \underline{\texttt{'9'}}$

Now *Number* is defined as either a digit or as a number followed by a digit:

def $Number : \mathcal{L}\ ASCII$
with $Number = Digit \cup Number \diamond Digit$

Note that *Number* is defined by a recursive equation. Because the type of all languages $\bigcup\mathcal{L}$ equipped with the subset relation $\subseteq$ is a complete lattice, the solution of such equations can be given explicitly as least fixed points. In the example above, for instance, we have that

$$\begin{array}{lll} Number & & \\ = & \{\textbf{def}\ Number\} & fix_{\mathcal{T},\subseteq}\ (A : \mathcal{L}\ ASCII\ .\ Digit \cup A \diamond Digit) \\ = & \{\textbf{def}\ fix\} & \bigcup(A : \mathcal{L}\ ASCII \mid A \subseteq Digit \cup A \diamond Digit) \end{array}$$

This fixed point exists because the function $(A : \mathcal{L}\ ASCII\ .\ Digit \cup A \diamond Digit)$ is continuous w.r.t. $\subseteq$. Further note that because $\mathcal{L}\ ASCII$ only contains finite sentences, the least fixed point solution is also the only solution of the equation.

5.2.3 Derived grammar operations

This section defines some additional frequently used operators in terms of the basic operators defined above.

The operator *Opt* is used to create *optional* languages. A language is called optional if it contains the empty sentence ε. Optional languages are therefore created by simply uniting the language with *Empty*:

spec $Opt : \bigcap(S : \mathcal{T}\ .\ \mathcal{L}\ S \circ\!\!\rightarrow \mathcal{L}\ S)$
with $\forall(A : \mathcal{L}\ \mathcal{U}\ .\ Opt\ A = A \cup Empty)$

Operator *One* is an injection operator from alphabets to languages. It maps a set of symbols to the language of sentences containing *one* symbol from the given set.

spec $One : \bigcap(S : \mathcal{T}\ .\ \mathcal{P}\ S \circ\!\!\rightarrow \mathcal{L}\ S)$
with $\forall(A : \mathcal{T}\ .\ One\ A = \{\tau \rceil A\})$

Note that $One\ A = \{a : A\ .\ \tau\ a\} = \bigcup(a : A\ .\ \underline{a})$.

The operators *Some* and *More* facilitate the definition of regular languages. *Some A* is the language of zero or more repetitions of sentences of A and *More A* is the language of one or more repetitions of sentences of A:

spec $Some, More : (\bigcap(S : \mathcal{T}\ .\ \mathcal{L}\ S \circ\!\!\rightarrow \mathcal{L}\ S))^2$
with $\forall(S : \mathcal{T}; A : \mathcal{L}\ S\ .\ Some\ A = Opt\ (More\ A) \wedge More\ A = A \diamond Some\ A)$

As in the *Number* example, *Some A* and *More A* can be given explicitly as least fixed points. Using *More* the definition of *Number* can be simplified to $Number = More\ Digit$.

5.3 Context sensitive languages

Using the operators defined so far, all context-free languages can be described. However, many languages, including Funmath itself, are *context-sensitive*. Mostly, this is due to the presence of declarations in the language. In Funmath, for instance, we have user defined operator patterns and operator precedence declarations. These declarations affect the way in which the expressions following the declarations are to be read. Therefore, Funmath is a context-sensitive language.

Two well known extensions of context-free grammars intended for the description of context-sensitive languages are attribute grammars [31, 1] and affix grammars [32]. Both formalisms extend context-free grammars by attaching values to the terminals and nonterminals, and providing computation rules for those values.

To illustrate the differences between both approaches we will express a simple context-sensitive, but not context-free language in both formalisms. This language consists of all sentences starting with a number of `a`s followed by the same number of `b`s and `c`s.

5.3.1 Attribute grammars

In attribute grammars, each context-free production rule can have a number of so called *attribution rules*. These rules, often formulated as statements in some programming language, define the computations on the attribute values. These values are denoted by attaching the attribute name to the corresponding nonterminal of the production rule. If the same nonterminal appears more than once in a production rule, then numbers are used to indicate which nonterminal of the production rule is meant. The following script shows how the `abc` example can be specified as an attribute grammar using the notation of [45], which also offers the possibility to include conditions which the attribute values must satisfy for the rule to be productive.

rule $abc ::= as\ bs\ cs\ .$
condition $as\ .\ n = bs\ .\ n$ **and** $bs\ .\ n = cs\ .\ n;$

rule $as ::= .$
attribution $as\ .\ n := 0$
rule $as ::= \texttt{'a'}\ as\ .$
attribution $as\ [0].n := 1 + as\ [1].n;$

rule $bs ::= .$
attribution $bs\ .\ n := 0$
rule $bs ::= \texttt{'b'}\ bs\ .$
attribution $bs\ [0].n := 1 + bs\ [1].n;$

rule cs ::= .
attribution cs . $n := 0$
rule cs ::= 'c' cs .
attribution cs $[0].n := 1 + cs$ $[1].n$;

The nonterminals as,bs and cs all have an attribute n, which contains the number of characters produced by the nonterminal. The attribution rules compute these values by initializing them with zero and then setting the attribute value of the left hand side nonterminal one plus the value of the right hand side nonterminal. The condition in the rule for abc finally ensures that the number of characters produced by the three nonterminals is the same, and therefore that abc indeed produces the desired language.

5.3.2 Affix grammars

Affix grammars are a member of the family of two-level grammars. These grammars all have in common that they have two levels of production rules:

1. The *meta level*. This level consist of *context-free* production rules. These rules use the symbol :: as production relation and are used to produce the *values* of the grammar. These values correspond to the attribute values in attribute grammars, which are usually values in some programming language. This constitutes one of the main differences between attribute grammars and two-level grammars: the attribute values are computed by functions outside the formalism, whereas values in two-level grammars are generated by production rules, which are part of the formalism.

2. The *hyper level*. The production relation : is used by the production rules of this level. The hyper rules are not context-free rules but rather schemes to generate a possibly infinite number of context-free rules. The generation is done by *consistent substitution* of the meta nonterminals appearing in the hyper rule, with values produced by those meta nonterminals. A substitution is called *consistent* if it substitutes the *same* value for every occurrence of a meta nonterminal in the hyper rule. This mechanism makes it possible to describe context-sensitive languages.

Affix grammars are a form of two-level grammars where the usage of meta nonterminals in hyper rules is restricted. Every hyper nonterminal has a fixed number of *affix positions*. Meta nonterminals are only allowed on those positions. Hyper nonterminals can be regarded as parametrized nonterminals and meta nonterminals as variables which can be used to construct the parameters. The fact that only consistent substitution of meta nonterminals is allowed, supports the view of meta nonterminals as variables. This is demonstrated by the following script which describes the `abc` example as an affix grammar, using Extended Affix Grammar [36, 18] notation:

```
zero :: "".
```

```
one :: "1".

n :: zero; one + n.

abc: as(n), bs(n), cs(n).

as(zero): .
as(one + n): "a", as(n).

bs(zero): .
bs(one + n): "b", bs(n).

cs(zero): .
cs(one + n): "c", cs(n).
```

The first three mata rules define a unary representation of natural numbers: the meta nonterminal `zero` represents the value 0 as the empty string, `one` represents the value 1 as the string `"1"`, and `n` can produce `zero` or the concatenation of `one` and `n`. The meta nonterminal `n` can therefore produce any finite sentence of ones.

The first hyper rule defines `abc` as the concatenation of `as(n)`, `bs(n)` and `cs(n)`. The second hyper rule states that `as("")` produces the empty string. Note that `zero` only produces `""`, so `as(""): .` is the only production generated by this rule. The third hyper rule states that `as(one+n)` produces `"a"` followed by `as(n)`. The meta nonterminal `one` only produces `"1"` and `n` can produce any finite sentence of ones, so one of the production rules generated by this hyper rule is `as("111"): "a", as("11").`. Now it is easy to see that `as` produces an `"a"` for each `1` of its affix. The hyper rules for `bs` and `cs` work in the same way.

Because only consistent substitution is allowed, the same production of `n` has to be substituted for `n` in the rule for `abc`. As a consequence, `abc` will only produce sentences having the same number of `as`, `bs` and `cs`. Hence, `abc` specifies the desired language.

5.4 Context-sensitive languages in Funmath

This section describes how context-sensitive languages are specified in Funmath using functions delivering languages. These *language functions* associate values with languages by mapping values to the corresponding language. This idea was introduced in Funmath in [38], although in a very different style using conditionals and tests. The approach presented below closely resembles the affix grammar approach with its affix expressions and hyper nonterminals (which can be seen as parametrized nonterminals), except that we will not require that the domain of the values is specified by a context-free grammar. In this respect, our approach looks more like attribute grammars, although there is no need to resort to another formalism for the description of the domains and computations: the domains of the values are Funmath types and the computations are Funmath functions.

The style we use to define language functions is the pattern matching style for the definition of functions as introduced in Section 4.7. The only difference is that we have to use the direct extension (see Section 4.3) of set union instead of function merge to join alternatives. The reason for this is that we want to be able to associate the same value with different languages, as is possible in affix grammars. An example of such an affix grammar is the following:

```
zero :: "".
one :: "1".
n :: zero; one + n.
m :: n

count_a(zero): .
count_a(zero): "b".
count_a(one): "a".
count_a(m + n): count_a(m), count_a(n).
```

The idea of this grammar is that `count_a(n)` produces strings containing `n` as and an arbitrary number of `b`s. The first three rules for `count_a` define the base cases: `zero` is associated with the empty language and the terminal `"b"` and `one` is associated with the terminal `"a"`. The final rule states that the number of `a`s in a concatenation of two parts is the sum of the number of `a`s in both parts. If we translate this grammar to Funmath using pattern matching we get the following definition:

$$
\begin{array}{l}
\textbf{def}\ \mathit{count\_a} : \mathbb{N} \to \mathcal{L}\ \mathit{ASCII} \\
\textbf{with}\ \mathit{count\_a} = \ 0 \mapsto \mathit{Empty}\ \& \\
\qquad\qquad\qquad\qquad\ \ 0 \mapsto \underline{\texttt{'b'}}\ \& \\
\qquad\qquad\qquad\qquad\ \ 1 \mapsto \underline{\texttt{'a'}}\ \& \\
\qquad\qquad\qquad\qquad\ \ \&(m : \mathbb{N}; n : \mathbb{N}\ .\ (m+n) \mapsto \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n)
\end{array}
$$

Clearly, this definition is incorrect, because there are incompatible alternatives. Even if we replace the first two alternatives by $0 \mapsto \mathit{Empty} \cup \underline{\texttt{'b'}}$, this is still incompatible with the last alternative. The replacement does show how this problem can be solved: if the same value is associated with different languages, then the language that has to be assigned to this value is the union of those different languages. This can be realized by using the direct extension of set union instead of function merge. In this way, the alternatives don't have to be compatible anymore. If the alternatives are compatible then we can of course still use the function merge operators. For *count_a* the use of set union results in:

$$
\begin{array}{l}
\textbf{def}\ \mathit{count\_a} : \mathcal{U} \to \mathcal{L}\ \mathit{ASCII} \\
\textbf{with}\ \mathit{count\_a} = \ 0 \mapsto \mathit{Empty}\ \hat{\bigcup} \\
\qquad\qquad\qquad\qquad\ \ 0 \mapsto \underline{\texttt{'b'}}\ \hat{\bigcup} \\
\qquad\qquad\qquad\qquad\ \ 1 \mapsto \underline{\texttt{'a'}}\ \hat{\bigcup} \\
\qquad\qquad\qquad\qquad\ \ \overset{<}{\bigcup}(m : \mathbb{N}; n : \mathbb{N}\ .\ (m+n) \mapsto \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n)
\end{array}
$$

We will elaborate the example *count_a* 0, which should be the language of sentences containing only bs:

$$
\begin{array}{ll}
 & \mathit{count\_a}\ 0 \\
= & \quad \{\, \textbf{def}\ \mathit{count\_a}\,\} \\
 & (\ 0 \mapsto \mathit{Empty}\ \hat{\bigcup}\ 0 \mapsto \underline{\texttt{'b'}}\ \hat{\bigcup}\ 1 \mapsto \underline{\texttt{'a'}}\ \hat{\bigcup} \\
 & \quad \overset{<}{\bigcup}(m : \mathbb{N}; n : \mathbb{N}\ .\ (m+n) \mapsto \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n))\ 0 \\
= & \quad \{\, \textbf{def}\ \hat{op}, \mapsto \,\} \\
 & \mathit{Empty} \cup \underline{\texttt{'b'}} \cup \overset{<}{\bigcup}(m : \mathbb{N}; n : \mathbb{N}\ .\ (m+n) \mapsto \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n)\ 0 \\
= & \quad \{\, \textbf{def}\ \overset{<}{op}\,\} \\
 & \mathit{Empty} \cup \underline{\texttt{'b'}} \cup (\bigcup \circ \mathbf{T}\ (m : \mathbb{N}; n : \mathbb{N}\ .\ (m+n) \mapsto \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n))\ 0 \\
= & \quad \{\, \textbf{def}\ \mathbf{T}, \mapsto \,\} \\
 & \mathit{Empty} \cup \underline{\texttt{'b'}} \cup (\bigcup \circ (k : \mathcal{U}\ .\ m : \mathbb{N}; n : \mathbb{N} \wedge k = m+n\ .\ \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n))\ 0 \\
= & \quad \{\, \textbf{def}\ \circ \,\} \\
 & \mathit{Empty} \cup \underline{\texttt{'b'}} \cup (k : \mathcal{U}\ .\ \bigcup(m : \mathbb{N}; n : \mathbb{N} \wedge k = m+n\ .\ \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n))\ 0 \\
= & \quad \{\, \beta\text{-reduction} \,\} \\
 & \mathit{Empty} \cup \underline{\texttt{'b'}} \cup \bigcup(m : \mathbb{N}; n : \mathbb{N} \wedge 0 = m+n\ .\ \mathit{count\_a}\ m \diamond \mathit{count\_a}\ n) \\
= & \quad \{\, m = n = 0 \,\} \\
 & \mathit{Empty} \cup \underline{\texttt{'b'}} \cup \mathit{count\_a}\ 0 \diamond \mathit{count\_a}\ 0
\end{array}
$$

Because languages only contain sentences of finite length, the result of this derivation, *count_a* 0 = *Empty* ∪ 'b' ∪ *count_a* 0 ◇ *count_a* 0, indeed specifies that *count_a* 0 contains all sentences containing only bs.

Note that *count_a* is also defined for all objects which are not natural numbers. For such objects *count_a* yields ∅. This is a result of the definition of direct extension and the strictness properties of ⋃, as described in Section 4.3. It may be desirable to have the more informative domain $\mathbb{N}$ for *count_a* instead of $\mathcal{U}$. In Section 5.6 we will define an operator with which we can control the domain of language functions.

The direct extension of the language concatenation operator ◇ is also very useful: with $\hat{\diamond}$ we can concatenate languages which are associated with the same value, which is convenient for describing context dependency. The abc language is an example of this:

def *abc* : $\mathcal{L}$ *ASCII*
with *abc* = ⋃(*as* $\hat{\diamond}$ *bs* $\hat{\diamond}$ *cs*)

The language function *as* $\hat{\diamond}$ *bs* $\hat{\diamond}$ *cs* associates *n* with *as* *n* ◇ *bs* *n* ◇ *cs* *n*. The abc language is the union of all these languages. The definitions of *as*, *bs* and *cs* can be derived directly from the affix grammar hyper rules given before:

$$\begin{array}{l}\textbf{def } as : \mathcal{U} \to \mathcal{L}\ ASCII \\ \textbf{with } as = \ 0 \mapsto Empty \mathbin{\hat{\bigcup}} \\ \qquad\qquad \overset{<}{\bigcup}(n : \mathbb{N} \,.\, (n+1) \mapsto \underline{\texttt{'a'}} \diamond as\ n)\end{array}$$

$$\begin{array}{l}\textbf{def } bs : \mathcal{U} \to \mathcal{L}\ ASCII \\ \textbf{with } bs = \ 0 \mapsto Empty \mathbin{\hat{\bigcup}} \\ \qquad\qquad \overset{<}{\bigcup}(n : \mathbb{N} \,.\, (n+1) \mapsto \underline{\texttt{'b'}} \diamond bs\ n)\end{array}$$

$$\begin{array}{l}\textbf{def } cs : \mathcal{U} \to \mathcal{L}\ ASCII \\ \textbf{with } cs = \ 0 \mapsto Empty \mathbin{\hat{\bigcup}} \\ \qquad\qquad \overset{<}{\bigcup}(n : \mathbb{N} \,.\, (n+1) \mapsto \underline{\texttt{'c'}} \diamond cs\ n)\end{array}$$

In the following example of the use of values we show how the *Number* language of Section 5.2.2 can be extended so that natural numbers are associated with their ASCII representations. First we define the language function *digit* which maps the first ten naturals to their representation:

$$\begin{array}{l}\textbf{def } digit : \mathcal{U} \to \mathcal{L}\ ASCII \\ \textbf{with } digit = \ 0 \mapsto \underline{\texttt{'0'}} \mathbin{\hat{\bigcup}} 1 \mapsto \underline{\texttt{'1'}} \mathbin{\hat{\bigcup}} 2 \mapsto \underline{\texttt{'2'}} \mathbin{\hat{\bigcup}} 3 \mapsto \underline{\texttt{'3'}} \mathbin{\hat{\bigcup}} 4 \mapsto \underline{\texttt{'4'}} \mathbin{\hat{\bigcup}} \\ \qquad\qquad 5 \mapsto \underline{\texttt{'5'}} \mathbin{\hat{\bigcup}} 6 \mapsto \underline{\texttt{'6'}} \mathbin{\hat{\bigcup}} 7 \mapsto \underline{\texttt{'7'}} \mathbin{\hat{\bigcup}} 8 \mapsto \underline{\texttt{'8'}} \mathbin{\hat{\bigcup}} 9 \mapsto \underline{\texttt{'9'}}\end{array}$$

It is not hard to see that $Digit = \bigcup digit$. We say that *Digit* is the language induced by the language function *digit*.

The language function *number* associates natural numbers with their representation:

$$\begin{array}{l}\textbf{def } number : \mathcal{U} \to \mathcal{L}\ ASCII \\ \textbf{with } number = \ digit \mathbin{\hat{\bigcup}} \\ \qquad\qquad \overset{<}{\bigcup}(n : \mathbb{N}; d : \square 10 \,.\, (10 \cdot n + d) \mapsto number\ n \diamond digit\ d)\end{array}$$

As with *digit*, we have that $Number = \bigcup number$, so the language induced by *number* is *Number*.

5.5 Defining iterating language functions

The following examples shows how the Funmath way of defining lists as functions defined on an initial part of $\mathbb{N}$, makes it possible to define iterating languages in a concise way without using recursion. An *iterating language* is a language that contains sentences which are concatenated from sentences of another language. An example of an iterating language is the language *numlist* of finite lists of numbers, which is an iteration of *number*. Using *Cat* we can define *numlist* by:

$$\textbf{def}\ numlist : \mathbb{N}^* \to \mathcal{L}\ ASCII$$
$$\textbf{with}\ numlist = (l : \mathbb{N}^*\ .\ Cat\ (i : \mathcal{D}\ l\ .\ (i = 0\ ?\ Empty \dagger \underline{\texttt{','}}) \diamond number\ (l\ i)))$$

The most simple recursive definition for *numlist* is:

$$\textbf{def}\ numlist : \mathcal{U} \to \mathcal{L}\ ASCII$$
$$\textbf{with}\ \ numlist =$$
$$\overset{<}{\bigcup}(\ m, n : \mathbb{N}^2; l : \mathbb{N}^*\ .$$
$$\varepsilon \mapsto Empty\ \hat{\bigcup}$$
$$\tau\ m \mapsto number\ m\ \hat{\bigcup}$$
$$(m \succ n \succ l) \mapsto number\ m \diamond \underline{\texttt{','}} \diamond numlist\ (n \succ l))$$

In other cases it can be even more cumbersome to replace *Cat* by recursion. An example of this is the following language function, which associates a bit string with the sentence containing an `a` for each 1 and a `b` for each 0 in the string:

$$\textbf{def}\ ab : \mathbb{B}^* \to \mathcal{L}\ ASCII$$
$$\textbf{with}\ ab = (l : \mathbb{B}^*\ .\ Cat\ (i : \mathcal{D}\ l\ .\ l\ i\ ?\ \underline{\texttt{'a'}} \dagger \underline{\texttt{'b'}}))$$

The shortest corresponding recursive version is:

$$\textbf{def}\ ab : \mathcal{U} \to \mathcal{L}\ ASCII$$
$$\textbf{with}\ ab = \overset{<}{\bigcup}(\ l, m : (\mathbb{B}^*)^2\ .$$
$$\varepsilon \mapsto Empty\ \hat{\bigcup}$$
$$\tau\ 1 \mapsto \underline{\texttt{'a'}}\ \hat{\bigcup}$$
$$\tau\ 0 \mapsto \underline{\texttt{'b'}}\ \hat{\bigcup}$$
$$(l +\!\!+ m) \mapsto ab\ l \diamond ab\ m)$$

These examples show that using *Cat* we can write much more concise language definitions for iterating languages than by using recursion. In Chapter 6 we will frequently take advantage of these possibilities of *Cat* in combination with lists.

5.6 Domains of language functions

Language functions constructed using direct extensions of $\bigcup$ all have domain $\mathcal{U}$. In some cases, it may be desirable to have a more informative domain. For this purpose we introduce the operator :: which maps a type and a language function to a language function which has the given type as domain:

$$\textbf{def}\ (— :: —) : \mathcal{T} \times Fam\ \mathcal{T} \to Fam\ \mathcal{T}$$
$$\textbf{with}\ \forall(A : \mathcal{T}; f : Fam\ \mathcal{T}\ .\ (A :: f) = (x : A\ .\ f\ x \dagger Fail))$$

Note that if the domain of the given language function is smaller than the given type, then the resulting language function returns *Fail* for the new domain values.

The operator is given precedence over equality and lower precedence than set union on the left and function merge on the right:

par $(— :: —) = (— :: —)$
par $(— \cup —) :: (— \,\&\, —)$

Using this relation we can define, for instance:

def $digit' : Fam\ (\mathcal{L}\ ASCII)$
with $digit' = \mathbb{N} ::$
$('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')$

For $n : \square 10$ this gives $digit'\ n = digit\ n$, and for all other natural numbers m we have $digit'\ m = Fail$.

Finally we introduce the notion *semantic domain*. The semantic domain of a language function is that part of its domain for which it doesn't yield *Fail*. The operator *semdom* yields the semantic domain of a given language function:

def $semdom : Fam\ \mathcal{T} \to \mathcal{T}$
with $\forall(F : Fam\ \mathcal{T}; x : \mathcal{U}\ .\ x \in semdom\ F \equiv x \in \mathcal{D}\ F \wedge F\ x \neq Fail)$

This operator satisfies

$$\forall(F : Fam\ \mathcal{T}; A : \mathcal{T}\ .\ semdom\ (A :: F) \subseteq A)$$

5.7 Semantic functions

In the previous sections we introduced language functions, mapping semantic values to sets of sentences to describe context-sensitive languages. A *semantic function* is a function which maps sentences of some language to semantic values. This notion is not as general as the notion of language function: with a semantic function it is impossible to associate a sentence with different semantic values, which is easy with language functions. This feature is indispensable for the description of *ambiguous* languages, in which the same sentence can have different meanings (semantic values). When describing unambiguous languages, the concept of semantic function is still very useful. In our approach, a language function is unambiguous iff it is disjoint, i.e. if all languages in its range are disjoint. We define disjointness by

def $disjoint : Fam\ \mathcal{T} \to \mathbb{B}$
with $\forall(F : Fam\ \mathcal{T}\ .\ disjoint\ F \equiv \forall(x : \bigcup F\ .\ \exists!(i : \mathcal{D}\ F\ .\ x \in F\ i)))$

We define the operator *semfun* so that for all disjoint language functions F, the function *semfun* F is the semantic function which maps each sentence from the induced language $\bigcup F$ to the unique semantic value in $\mathcal{D}\ F$ that is associated with the given sentence:

def $semfun : \bigtimes(F : \{\!|\, disjoint \,|\!\} \;.\; \bigcup F \to \mathcal{D}\ F)$
with $\forall(\ F : \{\!|\, disjoint \,|\!\}; s : \bigcup F\ .$
$\quad semfun\ F\ s = [i : \mathcal{D}\ F \mid s \in F\ i])$

A trivial example of a semantic function is the *identity* function on a language A. For every language A, this semantic function $id \rceil A$ maps a string from A to the string itself. The language function corresponding to this semantic function is $\iota \rceil A$ which associates every string with the language only containing that string. It is easy to derive that $disjoint\ (\iota \rceil A)$ holds and that $semfun\ (\iota \rceil A) = id \rceil A$.

Complicated semantic functions are also easy to specify using the operator *semfun*. We just define the corresponding language function, as demonstrated in the previous section, show that it is disjoint, and finally define the semantic function by applying *semfun* to the language function.

For example, if we want to define the semantic function which maps an ASCII representation of a natural number to the value it denotes, then we first define the language function which associates values with their representation. This is the function *number* defined in Section 5.4. Showing that *disjoint number* holds is an easy induction proof. Finally, the semantic function *numval* is defined by:

def $numval : Number \to \mathbb{N}$
with $numval = semfun\ number$

Thus, *numval* converts ASCII representations of numbers to their value, for instance $numval$ "`123`" $= 123$.

This way of defining semantic functions from language functions may look cumbersome, but in fact it is cleaner than defining semantic functions directly as functions over strings [10, 11], or alternatively as functions over parse trees [43]. The direct approaches need variables ranging over strings and parse trees respectively, whereas the language function approach only needs variables ranging over the semantic domain, which gives as better separation between syntax and semantics.

5.8 Removing ambiguity

In some cases, it is hard to specify an unambiguous language function for an unambiguous language. In such cases, it often is more convenient to first specify an ambiguous language function, and to resolve the ambiguities afterwards. There are many ways to eliminate ambiguities. One approach is to simply remove the sentences causing the ambiguities. This reduces the language induced by the ambiguous language function, which can decrease the expressivity of the language, and is therefore not always a good solution. Another more common approach is to *choose* one of the values associated with each ambiguous sentence. Below we give operators on language functions which support both approaches.

The operator *values* maps a language function F to another language function that associates each sentence of F with a family containing all values that are associated with that sentence by F:

def $values : Fam\ \mathcal{T} \to Fam\ \mathcal{T}$

with $\forall(F : Fam\ \mathcal{T}\ .\ values\ F = \overset{<}{\bigcup}(s : \bigcup F\ .\ (i : \mathcal{D}\ F \mid s \in F\ i) \mapsto \iota\ s))$

For every language function F, we have that $values\ F$ is unambiguous and that the language induced by $values\ F$ equals the language induced by F:

$$\forall(F : Fam\ \mathcal{T}\ .\ disjoint\ (values\ F) \wedge \bigcup(values\ F) = \bigcup F)$$

The operator *values* removes ambiguities by increasing the complexity of the semantic domain, which is illustrated by the following property:

$$semdom\ (values\ F) \subseteq \mathcal{D}\ F \rightharpoonup \mathcal{D}\ F$$

To be able to manipulate the semantic values of language functions, we define the *semantic computation* operator *semcomp* which maps a given function over the domain of a given language function:

def $semcomp : \mathcal{F} \to Fam\ \mathcal{T} \to Fam\ \mathcal{T}$

with $\forall(\ f : \mathcal{F}; F : Fam\ \mathcal{T}\ .$

$$semcomp\ f\ F = \overset{<}{\bigcup}(i : \mathcal{D}\ f \cap \mathcal{D}\ F\ .\ f\ i \mapsto F\ i))$$

For disjoint language functions we have:

$$\forall(F : \{\!|\ disjoint\ |\!\}\ .\ semfun\ (semcomp\ f\ F) = f \circ semfun\ F)$$

By using *choice functions* as semantic computations we can bring back the semantic domain of $values\ F$ to a subset of $\mathcal{D}\ F$. A choice function is a function which chooses a value from a given nonempty family. The type of all choice functions is therefore given by $\bigtimes(f : \mathcal{F} \setminus \iota\ \varepsilon\ .\ \{f\})$. Note that both choice operators $[\![]\!]$ and $[]$ are indeed members of this type.

If F is a language function and f is a choice function, then the language function $G := semcomp\ f\ (values\ F)$ has the following properties:

$$disjoint\ G \wedge semdom\ G \subseteq semdom\ F \wedge \forall(i : semdom\ G\ .\ G\ i \subseteq F\ i)$$

So G is unambiguous, its semantic domain is a subset of that of F and for each value i, the language associated with i by G is a subset of the language associated with i by F. Note that the last two properties also imply that the language induced by G is a subset of the language induced by F, i.e. $\bigcup G \subseteq \bigcup F$. Thus, we have produced an unambiguous language function G from an ambiguous language function F by using the choice function f. We introduce the operator *disambiguate*, so that $G = disambiguate\ f\ F$:

def $disambiguate : \mathcal{F} \to Fam\ \mathcal{T} \to Fam\ \mathcal{T}$

with $\forall(f : \mathcal{F}\ .\ disambiguate\ f = semcomp\ f \circ values)$

Note that the first argument of this operator need not be a choice function.

The different ways to resolve ambiguities can now be described as an application of *disambiguate* to the corresponding choice function. For instance, if we want to remove all ambiguous sentences from F, then we use *disambiguate* [] F. The language induced by the disambiguated language function only equals the language induced by F if F already was unambiguous:

$$\forall(F : Fam\ \mathcal{T}\ .\ disjoint\ F \equiv \bigcup F = \bigcup(disambiguate\ []\ F))$$

If we don't care which value is associated with an ambiguous sentence then we can use the deterministic choice operator $[\![]\!]$ and write $disambiguate[\![]\!]F$. The language induced by this disambiguation always equals the language induced by F:

$$\forall(F : Fam\ \mathcal{T}\ .\ \bigcup F = \bigcup(disambiguate[\![]\!]F))$$

Other choice functions that often occur in practice are *minimization* and *maximization* operators. This requires that the semantic domain is equipped with some ordering relation $\leq$ on some domain C. If this is the case, then *disambiguate* $min_{C,\leq}$ F associates each ambiguous sentence with the minimal value associated with that sentence. If there is no unique minimum element for the sentence, then it will be removed from the language. In the same way, *disambiguate* $max_{C,\leq}$ F associates each ambiguous sentence with the maximal value associated with that sentence.

5.9 Conclusions

We have defined operators to describe languages in Funmath, and introduced a theory of language functions, which can be used to describe the semantics of arbitrary formal languages, including ambiguous languages (languages which contain sentences having more than one meaning). The theory has been compared to the attribute grammar and affix grammar formalisms, which revealed notational similarities between language functions and affix grammars. We have compared language functions to semantic functions, and have defined an operator mapping unambiguous language functions to semantic functions. Finally, various ways to remove ambiguities from language functions have been described.

Chapter 6

A Syntax for Funmath

6.1 Introduction

One of the nice features of Funmath is that its basic expression syntax, as given in [16], is very simple. The syntax given in [16] already specifies most of the context-free part of the language of Funmath expressions and suggests by examples what the remaining syntactic features of the language are. To complete the syntax, we need to formally specify *variant applications* and give the syntactic precedence between the various syntactic forms of Funmath. Variant applications are applications of operators with affix conventions, such as prefix, postfix and infix operators. These affix conventions are defined by annotating the operator with dashes, which indicate the positions of the arguments, and are therefore also called *argument placeholders*. By introducing new operators with affix conventions, the user can extend the expression syntax dynamically. Therefore, this part of the syntax cannot be described with a context-free grammar. In this chaper, we will use the operators introduced in Chapter 5 to describe the complete Funmath syntax by language functions. We will also introduce an *operator precedence* mechanism and show how it is used to specify syntactic precedence between infix operators.

6.2 Lexical syntax

This section defines a lexical syntax for Funmath. The syntax is based on earlier variants described in [40] and [7].

6.2.1 ASCII Funmath

In this section we describe the Funmath lexical syntax as an ASCII-language. In the previous chapters, however, many symbols have been used that are not part of the ASCII character set. In our ASCII approach, these symbols are the result of *pretty printing*: an operator is just a list of ASCII strings. Each operator may have a pretty print form which specifies how the operator should be typeset. The following table gives possible ASCII

representations for some of the main operators:

$\mathcal{U}$	Univ	$\mathcal{D}$ —	Dom _	— ∨ ...	_ \/ ...	— · —	_ * _
$\mathcal{F}$	Func	— → —	_ -> _	— ∧ ...	_ /\ ...	—‾—	_ ^{ _ }
$\mathcal{T}$	Type	×...	>< ...	— ⇒ —	_ => _	— ̂ —	_ {^ _ } _
$\mathbb{B}$	Bin	⨯—	Cart	— ⇐ —	_ <= _	≤ —	{< _ }
$\mathbb{N}$	Nat	⋂—	Isect	— ≡ —	_ <=> _	—*	_ ^*
$\mathbb{Z}$	Int	— ⊆ —	_ subset _	∀—	All _		
$\mathbb{Q}$	Rat	— ∪ —	_ union _	∃—	Exist _		
$\mathbb{R}$	Real	— ∩ —	_ isect _	¬—	not _		
$\mathbb{C}$	Complex	ι —	iota _	∅	empty		

6.2.2 Predefined identifiers

Funmath has three types of predefined identifiers: numbers, characters and strings. The following definitions define the different numerical alphabets as sets of singleton strings:

def *Bit* := *One* {"01"}
def *OctDigit* := *One* {"01234567"}
def *Digit* := *One* {"0123456789"}
def *HexDigit* := *One* {"0123456789abcdefABCDEF"}

Note that *One* {"01"} = *One* {'0','1'} = {"0","1"}. The operator *numconst* generates the language of numbers over a given alphabet:

def *numconst* := $(A : \mathcal{L}\ \mathit{ASCII}\ .\ \mathit{More}\ A \diamond \mathit{Opt}\ (\underline{\text{'.'}} \diamond \mathit{More}\ A)$

The different types of numbers each have their own prefix. This prefix is optional for decimal numbers. This results in the following syntax for numerical constants:

def *NumConst* := {"0b","0B"} ◇ *numconst Bit* ∪
{"0q","0Q"} ◇ *numconst OctDigit* ∪
Opt ({"0d","0D"}) ◇ *numconst Digit* ∪
{"0x","0X"} ◇ *numconst HexDigit*

The following definition gives a C-like syntax with escape characters for ASCII characters:

def *Char* := *One* {" !#$%&()*+,-./0123456789:;<=>?"} ∪
One {"@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_"} ∪
One {"`abcdefghijklmnopqrstuvwxyz{|}~"} ∪
"\\" ◇ (*One* {"ntvbrfa\\\"\'\n"} ∪
OctDigit ◇ *OctDigit* ◇ *OctDigit* ∪
"x" ◇ *HexDigit* ◇ *HexDigit*)

Character constants are made by simply placing single quotes around one ASCII character:

def *CharConst* := '\'' ◇ *Char* ◇ '\''

String constants are made by placing double quotes around zero or more ASCII characters:

def *StringConst* := '\"' ◇ *Some Char* ◇ '\"'

The set of all predefined identifiers is the union of the three types of constants:

def *PredefId* := *NumConst* ∪ *CharConst* ∪ *StringConst*

6.2.3 Symbols

Funmath has several kinds of symbols: alphanumeric symbols, delimiters, special symbols and composed symbols. *Alphanumeric symbols* are made from letters digits and primes in the usual way:

def *Letter* := *One* {"abcdefghijklmnopqrstuvwxyz"} ∪
One {"ABCDEFGHIJKLMNOPQRSTUVWXYZ"}

def *AlphaSym* := *Letter* ◇ *Some* (*Letter* ∪ *Digit*) ◇ *Some* '\''

Delimiters are characters which form a symbol on their own, and can therefore not occur in any other symbol. Funmath has three delimiters:

def *Delimiter* := *One* {",()"}

A *special symbol* is composed of one or more special characters:

def *Special* := *One* {"!#$%&*+-./:;<=>?@[\\]^`{|}~"}

def *SpecSym* := *More Special*

And finally, a *composed symbol* is a symbol which is composed from alphanumeric and special symbols, by glueing them together with the underscore character:

def *CompSym* := (*AlphaSym* ∪ *SpecSym*) ◇ *More* ('_' ◇ (*AlphaSym* ∪ *SpecSym*))

The set of all symbols is now given by:

def *Symbol* := *AlphaSym* ∪ *Delimiter* ∪ *SpecSym* ∪ *CompSym* ∪ '_'

Funmath has the following keywords and placeholder symbols:

def *Keyword* := {"def","spec","par"}

def *Placeholder* := {"_","..."}

Furthermore, the following symbols are reserved for the construction of bindings and expressions:

def *Reserved* := {":",";","with","."} ∪ *Delimiter*

Keywords, placeholders and reserved symbols are not allowed as operator symbol. Therefore, the set of operator symbols is given by:

def *OpSymbol* := *Symbol* \ (*Keyword* ∪ *Placeholder* ∪ *Reserved*)

6.2.4 Layout

Symbols may (or sometimes must) be separated by layout. Layout is a combination of space and comments. Space is composed of one or more space, tab or newline characters:

def $Space := More\ (One\ \{\text{" \t\n"}\})$

Comments start with two underscore characters and are terminated with a newline character:

def $Comment := \underline{\text{"__"}} \diamond Some\ Char \diamond \underline{\text{"\n"}}$

Now, *Layout* can be defined by:

def $Layout := More\ (Space \cup Comment)$

6.2.5 Tokenization

A *token* is a predefined identifier or a symbol:

def $Token := PredefId \cup Symbol$

Tokenization means splitting up a string in tokens. This can be specified by the following language function, which maps a list of tokens and layout to its concatenation:

def $amb\_lex : (Token \cup Layout)^* \to \mathcal{L}\ ASCII$
with $amb\_lex = (toks : (Token \cup Layout)^*\ .\ Cat\ (\iota \circ toks))$

This language function is highly ambiguous because the concatenation of certain tokens can result in other valid tokens. This ambiguity is removed, as usual in lexical analysis, by picking the list starting with the longest prefix. For this purpose we define the following lexicographical ordering on lists of lists:

def $(— \ll —) : ((\mathcal{U}^*)^*)^2 \to \mathbb{B}$
with $\forall(\ a, b : (\mathcal{U}^*)^2; x, y : ((\mathcal{U}^*)^*)^2\ .$
$(\varepsilon \ll \varepsilon) \wedge$
$\neg(\varepsilon \ll (b \succ y)) \wedge$
$\neg((a \succ x) \ll \varepsilon) \wedge$
$((a \succ x) \ll (b \succ y)) \equiv (a = b\ ?\ (x \ll y)\ \dagger\ a \sqsubseteq b))$

Using the operator *disambiguate* (see Section 5.8), the operator $max_{(\mathcal{U}^*)^*,\ll}$ picks the desired list of tokens from a family of candidates. After this, the layout strings are removed:

def $lex : Token^* \to \mathcal{L}\ ASCII$
with $lex = [\ semcomp\ rm\_layout \circ disambiguate\ max_{(\mathcal{U}^*)^*,\ll}\ amb\_lex\ |$
$rm\_layout := (\ toks : (Token \cup Layout)^*\ .$
$cat\ (i : \mathcal{D}\ toks\ .\ toks\ i \in Token\ ?\ \tau\ toks\ i\ \dagger\ \varepsilon))]$

From now on, the Funmath syntax can be specified by languages of tokens instead of ASCII characters. This is because for any language function $f : A \to \mathcal{L}\ Token$, the function $p := (a : A\ .\ \bigcup(s : f\ a\ .\ lex\ s))$ is the language function of type $A \to \mathcal{L}\ ASCII$ corresponding to f. The function p can be seen as the composition of f with the lexical analyzer lex. So for every token language function we can construct the corresponding ASCII language function.

6.3 Funmath script syntax

A Funmath script is a list of Funmath declarations. There are three kinds of declarations:

Definitions: definitions are used to introduce identifiers for objects. It introduces an identifier for the object, which can also be a operator pattern, and binds an object to the identifier. The scope of a definition is global, which means that every free occurrence of an identifier in an expression refers to the value bound to that operator in its definition. It is not allowed to give more than one definition for the same identifier.

Specification: these also introduce objects, but don't guarantee uniqueness of the object. Giving a specification for an object that is already specified, results in *strengthening* of the object specification. That is, an occurrence of a specified object satisfies *all* specifications given for the object.

Precedence declarations: these are used to define syntactic precedence between infix and variadic operators. This is done using so called *precedence patterns*.

6.4 Operator patterns

6.4.1 Context-free part

This section gives a context-free grammar for operator patterns, which are composed of operator symbols and placeholder symbols. First we lift *OpSymbol*, which is an ASCII language, to a token language:

def $OpSym : \mathcal{L}\ Token$
with $OpSym = One\ OpSymbol$

Now we define the different kinds of operator patterns:

- Interior operators: operators whose pattern starts and ends with an operator symbol:

  def $InteriorOp : \mathcal{L}\ Token$
  with $InteriorOp = OpSym \diamond Some\ (Opt\ \underline{\text{"\_"}} \diamond OpSym)$

This includes exfix operators like `{ _ }` and operators without affix conventions like `Univ`.

- Prefix operators: operators whose pattern starts with an operator symbol and ends with —:

  def *PrefixOp* : $\mathcal{L}$ *Token*
  with *PrefixOp* = *InteriorOp* $\diamond$ "_"

- Postfix operators: operators whose pattern starts with — and ends with an operator symbol:

  def *PostfixOp* : $\mathcal{L}$ *Token*
  with *PostfixOp* = "_" $\diamond$ *InteriorOp*

- Infix operators: operators whose pattern starts and ends with the placeholder symbol `_` except function application:

  def *InfixOp* : $\mathcal{L}$ *Token*
  with *InfixOp* = "_" $\diamond$ *InteriorOp* $\diamond$ "_"

  Note that prefix, postfix and infix operators can also have interior arguments.

- Variadic operators: operators whose pattern is composed of one operator symbol followed by the placeholder symbol ...:

  def *VariadicOp* : $\mathcal{L}$ *Token*
  with *VariadicOp* = *OpSym* $\diamond$ "..."

  The best known variadic operator is the Cartesian product operator, which has `>< ...` as ASCII pattern. Variadic operators can have any number of arguments greater than one. These arguments are separated by the operator symbol. Note that the expression `A><B><C` is *not* a nested application: it is an application of the operator `>< ...` to the argument tuple `A,B,C`.

The language of all operators is now defined by:

def *Operator* : $\mathcal{L}$ *Token*
with *Operator* = *InteriorOp* $\cup$ *PrefixOp* $\cup$ *PostixOp* $\cup$ *InfixOp* $\cup$ *VariadicOp*

6.4.2 Context-sensitive part

There are two restrictions on the set of operators introduced by operator declarations and definitions:

Unique separation class

Every operator symbol must have the same *separation class* in every operator pattern it appears in. The separation class of a symbol is determined by two things:

1. whether the symbol is the first operator symbol of an operator pattern. These symbols will be called *opening symbols*.
2. whether the symbol is the last operator symbol of an operator pattern. These symbols will be called *closing symbols*.

The reason for this restriction is that it prevents ambiguities which can arise from operators having more than one operator symbol. An example of this is the *dangling else* problem, which occurred in early versions of the Algol syntax:

```
if _ then _
if _ then _ else _
```

If we now encounter the expression `if a then if b then c else d`, then we don't know to which `if` the `else` part belongs. Our restriction forbids this combination of patterns because the symbol `then` is a closing symbol in the first pattern but not in the second pattern. By adding an extra symbol to the first pattern, our restriction can be satisfied:

```
if _ then _ fi
if _ then _ else _
```

This combination of patterns does indeed not cause ambiguities. Another example of a legal set of operator patterns is:

```
not _
_ !
_ + _
>< ...
_ is similar to _
{ _ }
```

The operator symbols are divided in the separation classes mentioned above, as follows:

- opening and closing: `not, !, +, ><`
- only opening: `is, {`
- only closing: `to, }`
- neither opening nor closing: `similar`

We define the operator *opener*, which yields the opening symbol of an operator pattern by:

$$\begin{aligned}&\textbf{def}\ \mathit{opener} : \mathit{Operator} \to \mathit{OpSymbol}\\ &\textbf{with}\ \mathit{opener} = (\mathit{op} : \mathit{Variadic}\ \mathit{Op} \cup \mathit{InteriorOp} \cup \mathit{PrefixOp}\ .\ \alpha\ \mathit{op})\ \&\\ &\qquad\qquad (\mathit{op} : \mathit{InfixOp} \cup \mathit{PostfixOp}\ .\ \alpha\ \sigma\ \mathit{op})\end{aligned}$$

or alternatively, without using variables:

$$\begin{aligned}\mathit{opener} = {} & (\alpha \rceil (\mathit{Variadic}\ \mathit{Op} \cup \mathit{InteriorOp} \cup \mathit{PrefixOp}))\ \&\\ & ((\alpha \circ \sigma) \rceil (\mathit{InfixOp} \cup \mathit{PostfixOp}))\end{aligned}$$

The operator *closer* returns the closing symbol of a pattern:

$$\begin{aligned}&\textbf{def}\ \mathit{closer} : \mathit{Operator} \to \mathit{OpSymbol}\\ &\textbf{with}\ \mathit{closer} = (\mathit{op} : \mathit{Variadic}\ \mathit{Op}\ .\ \alpha\ \mathit{op})\ \&\\ &\qquad\qquad (\mathit{op} : \mathit{InteriorOp} \cup \mathit{PostfixOp}\ .\ \omega\ \mathit{op})\ \&\\ &\qquad\qquad (\mathit{op} : \mathit{InfixOp} \cup \mathit{PrefixOp}\ .\ \omega\ \rho\ \mathit{op})\end{aligned}$$

The separation class of a token *tok* of an operator pattern *op* is now given by the pair $(\mathit{tok} = \mathit{opener}\ \mathit{op}, \mathit{tok} = \mathit{closer}\ \mathit{op})$. A family of operators whose operator symbols all are in the same separation class for every operator in the family, will be called *separable*:

$$\begin{aligned}&\textbf{def}\ \mathit{separable} : \mathit{Fam}\ \mathit{Operator} \to \mathbb{B}\\ &\textbf{with}\ \ \mathit{separable} = (\mathit{ops} : \mathit{Fam}\ \mathit{Operator}\ .\\ &\qquad \forall(\ \mathit{tok} : \mathit{OpSymbol}\ .\\ &\qquad\quad \mathit{con}\ (\mathit{op} : \{\mathit{ops}\} \mathbin{\mathbb{A}} \mathit{tok} \in \{\mathit{op}\}\ .\ (\mathit{tok} = \mathit{opener}\ \mathit{op}, \mathit{tok} = \mathit{closer}\ \mathit{op}))))\end{aligned}$$

The restriction that every operator symbol must have the same separation class in every operator pattern from *ops* it appears in, can now be expressed by *separable ops*. For a separable family of operators *ops*, we can therefore define the function *class ops* which, given a token appearing in one of the operator patterns, yields the separation class of the token:

$$\begin{aligned}&\textbf{def}\ \mathit{class} : \{\!|\, \mathit{separable} \,|\!\} \to \mathit{OpSymbol} \rightharpoonup \mathbb{B}^2\\ &\textbf{with}\ \mathit{class} = (\ \mathit{ops} : \{\!|\, \mathit{separable} \,|\!\}\ .\ \&(\mathit{op} : \{\mathit{ops}\}; \mathit{tok} : \mathit{OpSymbol} \cap \{\mathit{op}\}\ .\\ &\qquad\qquad \mathit{tok} \mapsto (\mathit{tok} = \mathit{opener}\ \mathit{op}, \mathit{tok} = \mathit{closer}\ \mathit{op})))\end{aligned}$$

Unique placeholder structure

The other restriction on operator patterns is that it must be possible to identify each operator pattern by its sequence of operator symbols. That is, different operators must have different sequences of operator symbols. More formally, the function that removes the placeholder symbols must be injective on the given family of operators:

$$\begin{aligned}&\textbf{def}\ \mathit{unique\_placeholders} : \mathit{Fam}\ \mathit{Operator} \to \mathbb{B}\\ &\textbf{with}\ \ \mathit{unique\_placeholders} = (\mathit{ops} : \mathit{Fam}\ \mathit{Operator}\ .\\ &\qquad \mathit{inj}\ (\mathit{op} : \{\mathit{ops}\}\ .\ \mathit{cat}\ (i : \mathcal{D}\ \mathit{op}\ .\ \mathit{op}\ i \in \mathit{Placeholder}\ ?\ \varepsilon \dagger \tau\ \mathit{op}\ i)))\end{aligned}$$

The reason for this restriction is that it prevents ambiguities caused by operators using the same operator symbols, but having a different fixity. For instance, if we have the patterns

```
[ _ ]
_ [ _ ] _
```

then the expression `a[b]c` can be parsed both as a nested function application of the first pattern with argument `b`, as in `(a[b])c`, and as an application of the second pattern with three arguments. Another problem with such patterns is partial application, which is done by omitting arguments. For operators having the same operator symbols it may not be possible to tell which operator is meant in a partial application.

The predicate *valid_ops* combines both restrictions on operator patterns:

def $valid\_ops : Fam\ Operator \to \mathbb{B}$
with $valid\_ops = separable \mathbin{\hat{\wedge}} unique\_placeholders$

6.5 Precedence patterns

Precedence declarations are used to define the syntactic precedence between individual infix and variadic operators. Operator precedence is introduced in the form of *operator precedence patterns*. An operator precedence pattern basically is an application containing *only* infix and variadic operators, in which all operator arguments are surrounded by parentheses. The idea is that this parenthesized application shows how applications without parenthesis should be parsed. For this reason, the patterns are also called *parenthesis insertion patterns*.

A simple example of such patterns is `(_*_)+_` which specifies that all applications of `*` are valid left arguments of `+`. This results in `a*b+c` being parsed as `(a*b)+c`, which is the only parsing that fits in the precedence pattern. The placeholder symbols in the patterns are optional, so above we could also have written `(*)+`. Precedence patterns can also be used to express self association for infix operators: the pattern `(_+_)+_` specifies that `+` associates to the left.

Patterns can have two arguments, like `(_*_)+(_*_)`, which additionally specifies that applications of `*` are also valid right arguments of `+`. It even is possible to have nested precedence patterns like `((^)*(^))+(*)`, which combines the patterns `(^)*(^)` and `(*)+(*)`.

The exact meaning of operator patterns is obtained by translating them into so called *operator precedence relations*. This is a pair of two relations on operators, which me usually denote by the operators $— \prec —$ and $— \succ —$, whose precedence is given by:

par $(— \prec —) \wedge (— \prec —)$
par $(— \succ —) \wedge (— \succ —)$

The intention of the precedence relations $\prec, \succ : (Operator^2 \to \mathbb{B})^2$ is that $op \prec op'$ holds iff applications of op are valid *left* arguments of op', and $op \succ op'$ holds iff applications of op' are valid *right* arguments of op. These relations resemble the precedence relations derived

from operator precedence grammars as described in [24]. Using precedence relations to *specify* syntactic operator precedence in Funmath was proposed in [39].

6.5.1 Context-free part

First we define the auxiliary operator *interior* which strips the outer placeholders from an operator pattern:

$$
\begin{array}{l}
\textbf{def}\ \mathit{interior} : \mathit{Operator} \to \mathit{OpSymbol}^* \\
\textbf{with}\ \mathit{interior} = \ (\mathit{op} : \mathit{InteriorOp}\ .\ \mathit{op})\ \& \\
\qquad\qquad\qquad\quad (\mathit{op} : \mathit{PostfixOp}\ .\ \sigma\ \mathit{op})\ \& \\
\qquad\qquad\qquad\quad (\mathit{op} : \mathit{PrefixOp} \cup \mathit{VariadicOp}\ .\ \rho\ \mathit{op})\ \& \\
\qquad\qquad\qquad\quad (\mathit{op} : \mathit{InfixOp}\ .\ \rho\ \sigma\ \mathit{op})
\end{array}
$$

The syntax for precedence patterns is given by:

$$
\begin{array}{l}
\textbf{def}\ \mathit{prec\_pat}, \mathit{prec\_arg}, \mathit{prec\_int} : (\mathit{Fam}\ (\mathcal{L}\ \mathit{Token}))^* \\
\textbf{with}\ \ \mathit{prec\_pat} = \mathcal{P}\ \mathit{Operator} \times (\mathit{Operator}^2 \to \mathbb{B})^2 :: \\
\qquad \overset{<}{\bigcup}(\ \mathit{lroot}, \mathit{rroot} : (\mathcal{P}\ \mathit{Operator})^2; \mathit{op} : \mathit{InfixOp} \cup \mathit{VariadicOp}; \\
\qquad\quad LL, LR, RL, RR, \prec, \succ : (\mathit{Operator}^2 \to \mathbb{B})^6 \wedge \\
\qquad\quad \forall (a, b : \mathit{Operator}^2\ .\ \ (a \prec b \equiv a\langle LL\rangle b \vee a\langle RL\rangle b \vee a \in \mathit{lroot} \wedge b = \mathit{op}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (a \succ b \equiv a\langle LR\rangle b \vee a\langle RR\rangle b \vee a = \mathit{op} \wedge b \in \mathit{rroot}))\ . \\
\qquad\quad (\iota\ \mathit{op}, (\prec, \succ)) \mapsto \\
\qquad\quad \mathit{prec\_arg}\ (\mathit{lroot}, (LL, LR)) \diamond \mathit{prec\_int}\ \mathit{op} \diamond \mathit{prec\_arg}\ (\mathit{rroot}, (RL, RR)))
\end{array}
$$

The language function *prec_pat* has two arguments. The first argument contains the root operator of the precedence pattern and the second argument contains the precedence relations produced by the precedence pattern. The relations $(\prec, \succ)$, produced by a precedence pattern are constructed by taking the disjunction of the relations produced by the arguments and the relations which state that the root operator of the left argument is a valid left argument of the root operator of the pattern and that the root operator of the right argument is a valid right argument of the root operator of the pattern. The syntax of precedence arguments and precedence interiors is given by:

$$
\begin{array}{l}
\wedge\ \mathit{prec\_arg} = \\
\quad (\emptyset, (0^\bullet \mathit{Operator}^2)^2) \mapsto \mathit{Opt}\ \underline{\text{"\_"}}\ \hat{\bigcup} \\
\quad (x : \mathcal{D}\ \mathit{prec\_pat}\ .\ \underline{\text{"("}} \diamond \mathit{prec\_pat}\ x\ \underline{\text{")"}}) \\
\wedge\ \mathit{prec\_int} = \\
\quad (\ \mathit{op} : \mathit{Operator}\ .\ [\mathit{iop} := \mathit{interior}\ \mathit{op}\ . \\
\qquad \mathit{Cat}\ (i : \mathcal{D}\ \mathit{iop}\ .\ \mathit{iop}\ i \in \mathit{Placeholder}\ ?\ \mathit{Opt}\ \underline{\mathit{iop}\ i} \dagger \underline{\mathit{iop}\ i})])
\end{array}
$$

A precedence argument is either an optional placeholder symbol, in which case the argument has no root operator and produces no precedences, or a precedence pattern surrounded by parentheses, in which case the arguments are just passed. A precedence interior is an interior in which the placeholders are optional.

6.5.2 Context-sensitive part

The first restriction on the precedence relations is obvious: it is not allowed for one operator to be a left argument of another operator while the latter operator is a right argument of the first operator. For instance, if we have the patterns `(_*_)+_` and `_*(_+_)`, then the first pattern says that `a*b+c` must be parsed as `(a*b)+c`, while the latter says that it should be parsed as `a*(b+c)` In terms of precedence relations this restriction means that $op \prec op' \wedge op \succ op'$ is not allowed, so the conjunction of both precedence relations must yield false for every pair of operators:

def $unamb\_rels : (Operator^2 \to \mathbb{B})^2 \to \mathbb{B}$
with $\forall(L, R : Operator^2 \to \mathbb{B} \;.\; unamb\_rels\ (L, R) \equiv \neg\exists(L\ \hat{\wedge}\ R))$

The restriction that at most one precedence relation may hold between two operators is also present in operator precedence grammars [24].

This restriction alone is not sufficient, however. It is still possible that ambiguities arise in nested applications. For instance, if we have the unambiguous list of patterns

```
par (_*_)+_
par _*(_^_)
par _^(_+_)
```

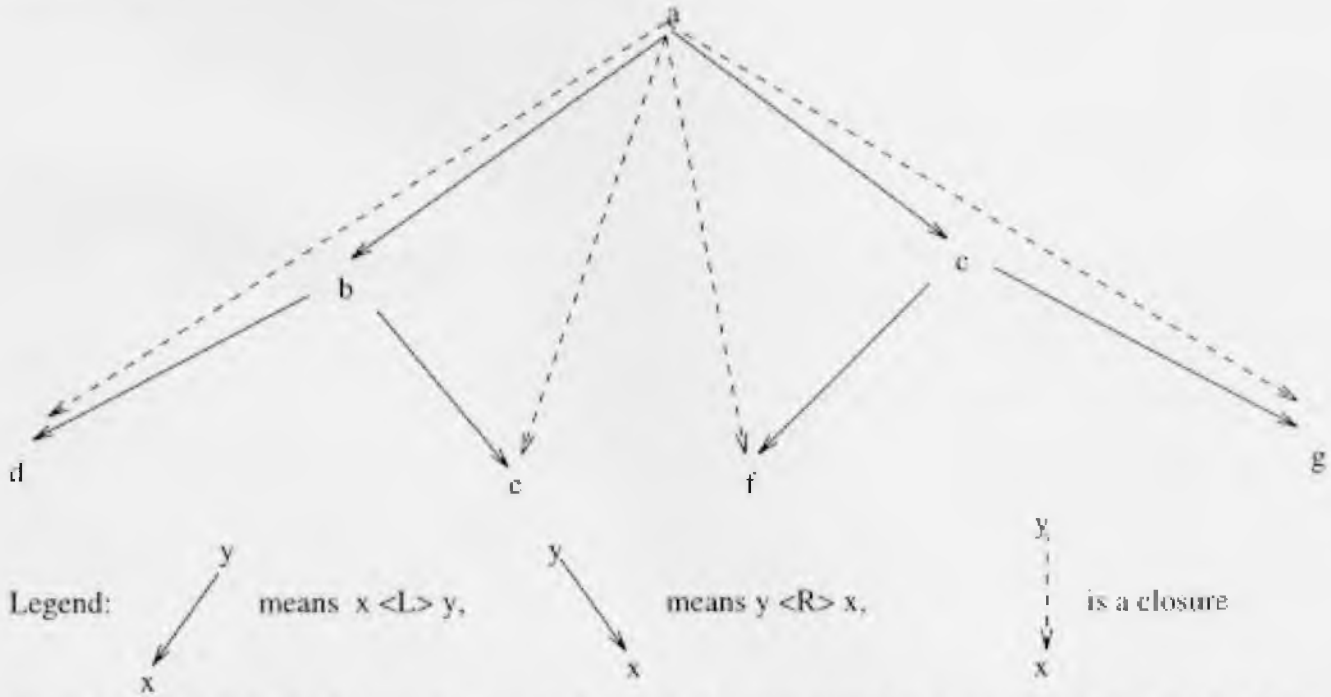
then the expression `a*b^c+d` can be parsed as `(a*(b^c))+d`, but also as `a*(b^(c+d))`. These ambiguities can be prevented by first performing a transitive closure operation on the precedence relations, and then checking for ambiguities in the closed relations. The closure operation that has to be performed is the following: if operator $op1$ is a left argument of $op2$ then *all* arguments of $op1$ must also be allowed as left arguments of $op2$, similarly, if $op1$ is a right argument of $op0$, then *all* arguments of $op1$ must be allowed as right arguments of $op0$. In terms of precedence relations, this closure can be expressed as a least fixed point w.r.t. the following ordering relation on pairs of precedence relations:

def $PRORD : ((Operator^2 \to \mathbb{B})^2)^2 \to \mathbb{B}$
with $\forall(\ L0, R0, L1, R1 : (Operator^2 \to \mathbb{B})^4 \;.$
$(L0, R0)\langle PRORD\rangle(L1, R1) \equiv \forall(L0 \Rightarrow L1) \wedge \forall(R0 \Rightarrow R1))$

Now we can define the closure by:

$$
\begin{array}{l}
\textbf{def}\ close : (Operator^2 \to \mathbb{B})^2 \to (Operator^2 \to \mathbb{B})^2 \\
\textbf{with}\ \ close = (L, R : (Operator^2 \to \mathbb{B})^2 \ . \\
\qquad fix_{(Operator^2 \to \mathbb{B})^2, PRORD}\ (\prec, \succ : (Operator^2 \to \mathbb{B})^2 \ . \\
\qquad (\ op0, op2 : Operator^2 \ .\ op0 \langle L \rangle op2 \vee \\
\qquad\quad \exists (op1 : Operator \ .\ (op0 \prec op1 \vee op1 \succ op0) \wedge op1 \prec op2)), \\
\qquad (\ op0, op2 : Operator^2 \ .\ op0 \langle R \rangle op2 \vee \\
\qquad\quad \exists (op1 : Operator \ .\ op0 \succ op1 \wedge (op1 \succ op2 \vee op2 \prec op1)))))
\end{array}
$$

Taking the least fixed point w.r.t. *PRORD* makes sure that the *smallest* closure is taken, that is, the closure whose relations contain the smallest number of ones. The four closure rules contained in the existential quantifications of the definition can be represented pictorially as follows:



The closure of the relations generated by the three precedence patterns, given in the example before, also contains (_^_)+_ which is inconsistent with the third pattern _^(_+_). So this combination of patterns will rightly be rejected.

The closure rules presented above were first introduced in [39] together with a proof that the rules combined with the unambiguity restriction indeed guarantee that infix expressions can be parsed in just one way. A parsing algorithm which uses the precedence relations to parse infix expressions in quadratic time and linear space will be presented in Chapter 7.

6.6 Expressions

This section defines the main part of the Funmath syntax: the expression syntax. Before giving the syntax, we first define the representation of Funmath expressions. This repre-

sentation is not only important for understanding the meaning of expressions, but also for future *tools* supporting the Funmath language, which can use the representation as data structure for Funmath expressions. The representation is therefore designed for optimal manipulability by tools, whereas the syntax is designed for optimal readability by humans. As a result, the representation has fewer constructs than the expression syntax. This is especially true for applications; the syntax offers many different kinds of applications (default, prefix, postfix, infix and variadic applications), whereas the representation only has one construct for application.

6.6.1 Funmath representation

We first introduce the language of *internal operator identifiers*. These identifiers are not part of *Operator* and can therefore not be introduced by users. As a consequence, these identifiers can safely be used by tools, without having to worry about clashes with identifiers introduced by the user. An internal identifier starts with an underscore followed by an alphanumeric symbol:

def $InternalOp := One\ ('\_' \diamond AlphaSym)$

The language $Operator'$ contains all internal and regular operators:

def $Operator' := Operator \cup InternalOp$

Now we define the representations of expressions and bindings as recursive data types, whose first component is a string which indicates the type of the representation node:

def $Expr : \mathcal{T}; Binding : \mathcal{T}$
with $Expr \times Binding =$
$fix_{\mathcal{T},\subseteq}\ (\&(E, B : \mathcal{T}^2\ .\ (E \times B) \mapsto$
$(\ \iota\ \tau\ \text{"}\texttt{empty}\text{"} \cup$
$\iota\ \text{"}\texttt{const}\text{"} \times PredefId \cup$
$\iota\ \text{"}\texttt{op}\text{"} \times Operator' \cup$
$\{\text{"}\texttt{appl}\text{"}, \text{"}\texttt{vappl}\text{"}\} \times E \times E \cup$
$\iota\ \text{"}\texttt{tup}\text{"} \times E^* \cup$
$\{\text{"}\texttt{abstr}\text{"}, \text{"}\texttt{vtabstr0}\text{"}, \text{"}\texttt{vtabstr1}\text{"}, \text{"}\texttt{partap}\text{"}\} \times B \times E)$
$\times$
$(\ \iota\ \text{"}\texttt{var}\text{"} \times Operator' \cup$
$\iota\ \text{"}\texttt{btup}\text{"} \times B^* \cup$
$\{\text{"}\texttt{type}\text{"}, \text{"}\texttt{assign}\text{"}\} \times B \times E \cup$
$\iota\ \text{"}\texttt{filter}\text{"} \times B \times E)))$

A *context representation* has five components. The first component is the list of declared identifiers, the second component contains the precedence relations generated by precedence declarations, the third component contains the bindings introduced by definitions and the fourth the bindings introduced by specifications. Inside expressions, the fifth component contains the local context, i.e. the binders and filters of surrounding abstractions. Note that the first two components of the context make Funmath a context sensitive language. They greatly influence the expression syntax, while the other compenents have no influence on the form of the expressions at all.

$$\textbf{def}\ \mathit{Context} :=$$
$$\mathit{Operator}^* \times (\mathit{Operator}^2 \to \mathbb{B})^2 \times \mathit{Binding}^* \times \mathit{Binding}^* \times (\mathit{Binding} \cup \mathit{Expr})^*$$

To define the access functions on *Context* we first define the *generic get* and *set* operators, which given a product type and an index in that type, yield the read and write function for that index in the type respectively:

$$\textbf{def}\ \mathit{generic\_get} : \bigcap(F : \mathit{Fam}\ \mathcal{T}; i : \mathcal{D}\ F\ .\ \iota \bigtimes F \times \iota\ i \circ\!\!\to \bigtimes F \to F\ i)$$
$$\textbf{with}\ \mathit{generic\_get} = \&(F : \mathit{Fam}\ \mathcal{T}; i : \mathcal{D}\ F\ .\ (\bigtimes F, i) \mapsto (t : \bigtimes F\ .\ t\ i))$$

$$\textbf{def}\ \mathit{generic\_set} : \bigcap(F : \mathit{Fam}\ \mathcal{T}; i : \mathcal{D}\ F\ .\ \iota \bigtimes F \times \iota\ i \circ\!\!\to \bigtimes F \times F\ i \to \bigtimes F)$$
$$\textbf{with}\ \mathit{generic\_set} =$$
$$\&(F : \mathit{Fam}\ \mathcal{T}; i : \mathcal{D}\ F\ .\ (\bigtimes F, i) \mapsto (t : \bigtimes F; x : F\ i\ .\ t \triangleleft (i \mapsto x)))$$

Now the read and write functions on the different components of *Context* are defined by applying the generic operators to *Context* and the corresponding index:

$$\textbf{def}\ \mathit{get\_ops} := \mathit{generic\_get}\ (\mathit{Context}, 0); \mathit{set\_ops} := \mathit{generic\_set}\ (\mathit{Context}, 0)$$
$$\textbf{def}\ \mathit{get\_rels} := \mathit{generic\_get}\ (\mathit{Context}, 1); \mathit{set\_rels} := \mathit{generic\_set}\ (\mathit{Context}, 1)$$
$$\textbf{def}\ \mathit{get\_defs} := \mathit{generic\_get}\ (\mathit{Context}, 2); \mathit{set\_defs} := \mathit{generic\_set}\ (\mathit{Context}, 2)$$
$$\textbf{def}\ \mathit{get\_specs} := \mathit{generic\_get}\ (\mathit{Context}, 3); \mathit{set\_specs} := \mathit{generic\_set}\ (\mathit{Context}, 3)$$
$$\textbf{def}\ \mathit{get\_loc} := \mathit{generic\_get}\ (\mathit{Context}, 4); \mathit{set\_loc} := \mathit{generic\_set}\ (\mathit{Context}, 4)$$

The operator *vartree* takes a binding representation and returns the representation of the tuple containing all operators bound by the binding:

$$\textbf{def}\ \mathit{vartree} : \mathit{Binding} \to \mathit{Expr}$$
$$\textbf{with}\ \mathit{vartree} =$$
$$\&(\ op : \mathit{Operator}'; bs : \mathit{Binding}^*; b : \mathit{Binding}; e : \mathit{Expr}$$
$$(\text{"}\texttt{var}\text{"}, op) \mapsto (\text{"}\texttt{op}\text{"}, op)\ \&$$
$$(\text{"}\texttt{btup}\text{"}, bs) \mapsto (\text{"}\texttt{tup}\text{"}, \mathit{vartree} \circ bs)\ \&$$
$$(\text{"}\texttt{type}\text{"}, b, e) \mapsto \mathit{vartree}\ b\ \&$$
$$(\text{"}\texttt{assign}\text{"}, b, e) \mapsto \mathit{vartree}\ b\ \&$$
$$(\text{"}\texttt{filter}\text{"}, b, e) \mapsto \mathit{vartree}\ b)$$

The operator *univar* maps an operator to the binding that binds the given operator to the universal type $\mathcal{U}$:

$$\begin{array}{l}\textbf{def}\ \mathit{univar} : \mathit{Operator}' \to \mathit{Binding}\\ \textbf{with}\ \mathit{univar} = (\mathit{op} : \mathit{Operator}'\ .\ \text{"}\texttt{type}\text{"}, (\text{"}\texttt{var}\text{"}, \mathit{op}), (\text{"}\texttt{op}\text{"}, \tau\ \text{"}\texttt{\_Univ}\text{"}))\end{array}$$

The operator *print* converts a natural number to its ASCII representation:

$$\begin{array}{l}\textbf{def}\ \mathit{print} : \mathbb{N} \to \mathit{ASCII}^*\\ \textbf{with}\ \mathit{print} = \ (\text{"}\texttt{0}\text{"}, \text{"}\texttt{1}\text{"}, \text{"}\texttt{2}\text{"}, \text{"}\texttt{3}\text{"}, \text{"}\texttt{4}\text{"}, \text{"}\texttt{5}\text{"}, \text{"}\texttt{6}\text{"}, \text{"}\texttt{7}\text{"}, \text{"}\texttt{8}\text{"}, \text{"}\texttt{9}\text{"})\ \&\\ \qquad\qquad \&(n : \mathbb{N}; d : \square 10 \wedge\ n > 0\ .\ (10 \cdot n + d) \mapsto \mathit{print}\ n +\!\!+ \mathit{print}\ d)\end{array}$$

These operators will be used by the operator *appl_node*, which maps an operator and an argument list to the representation of the application of that operator to those arguments. The operator *appl_node* also takes care of the representation of partial applications. Partial applications are applications in which some arguments are omitted (see Section 2.2.4). These missing arguments are represented by the node τ "`empty`".

$$\begin{array}{l}\textbf{def}\ \mathit{appl\_node} : \mathit{Operator}' \times \mathit{Expr}^* \to \mathit{Expr}\\ \textbf{with}\ \ \mathit{appl\_node} =\\ \qquad \&(\ \mathit{op} : \mathit{Operator}'; n : \mathbb{N}; \mathit{es} : \mathit{Expr}^n; I := \{i : \square n \mid \mathit{es}\ i = \tau\ \text{"}\texttt{empty}\text{"}\};\\ \qquad\quad (\mathit{op}, \mathit{es}) \mapsto\\ \qquad\quad (\ I = \square n\ ?\ (\text{"}\texttt{op}\text{"}, \mathit{op})\ \dagger\\ \qquad\qquad I = \emptyset \wedge n = 1\ ?\ (\text{"}\texttt{vappl}\text{"}, (\text{"}\texttt{op}\text{"}, \mathit{op}), \alpha\ \mathit{es})\ \dagger\\ \qquad\qquad I = \emptyset\ ?\ (\text{"}\texttt{vappl}\text{"}, (\text{"}\texttt{op}\text{"}, \mathit{op}), (\text{"}\texttt{tup}\text{"}, \mathit{es}))\ \dagger\\ \qquad\qquad [\ \mathit{ops} := (i : I\ .\ \tau\ (\text{"}\texttt{\_x}\text{"} +\!\!+ \mathit{print}\ i));\\ \qquad\qquad\quad e := \mathit{appl\_node}\ (\mathit{op}, \mathit{es} \triangleleft (i : I\ .\ \text{"}\texttt{op}\text{"}, \mathit{ops}\ i));\\ \qquad\qquad\quad b := \ \#I = 1\ ?\ \mathit{univar}\ [\mathit{ops}]\ \dagger\\ \qquad\qquad\qquad\qquad (\text{"}\texttt{btup}\text{"}, \mathit{cat}\ (i : \square n\ .\ i \in I\ ?\ \tau\ \mathit{univar}\ (\mathit{ops}\ i)\ \dagger\ \varepsilon))\ .\\ \qquad\qquad\quad \text{"}\texttt{partap}\text{"}, b, e]))\end{array}$$

If all arguments are omitted, then *appl_node* just returns an operator node. If the operator only has one argument and the argument is not empty, then *appl_node* returns the application node with the operator as function and that argument as argument. If there is more than one argument and all arguments are not empty, then *appl_node* yields the application node which applies the operator to the tuple of all arguments. The final case of *appl_node* handles partial applications. It generates an internal identifier for each empty argument, and creates the application node with all empty arguments replaced by their identifiers. Then it creates a binding node which binds all new identifiers to the universal type. Finally, a partial application node containing the generated binding and application is returned. In this way, nodes produced by *appl_node* will never contain empty arguments.

6.6.2 Grammar

In [16] a simple context-free grammar for *binding* and *expr* is specified. However, that grammar is ambiguous as it doesn't specify the precedence between the various syntactic constructs. Furthermore, it doesn't specify the form of variant applications. The following definition is an extension of the grammar in [16] giving the complete Funmath syntax:

$$\begin{aligned}\textbf{def}\ & int\_appl, int\_arg, unit, default\_appl, default\_appl', prefix\_appl, prefix\_appl',\\ & prefix\_arg, postfix\_appl, postfix\_appl', postfix\_arg, infix\_appl, infix\_appl',\\ & tuple, tuple', abstraction, abstraction',\\ & expr, op\_appl, left\_arg, right\_arg, varunit, vartuple, vartuple'\\ & binder, binder', bindtuple, bindtuple', binding : (Fam\ (\mathcal{L}\ Token))^* \ \textbf{with}\end{aligned}$$

The definition header gives the identifiers of all language functions that the grammar is composed of. Because all language functions depend on each other, it is impossible to define them in separate definitions.

The language function *int_appl* produces interiors of applications, and *int_arg* produces interior arguments:

$$\begin{aligned}& int\_appl = Operator \times Context \times Expr^* ::\\ & \overset{<}{\bigcup}(\ op : Operator; iop := interior\ op; c : Context; args : Expr^{\#iop}\ .\\ & \quad (op, c, cat\ (i : \mathcal{D}\ iop\ .\ iop\ i = \text{"\_"}\ ?\ \tau\ args\ i \dagger \varepsilon)) \mapsto\\ & \quad Cat\ (i : \mathcal{D}\ iop\ .\ iop\ i = \text{"\_"}\ ?\ int\_arg\ (c, args\ i) \dagger \underline{iop\ i}))\\ & \wedge int\_arg = \overset{<}{\bigcup}(c : Context\ .\ (c, \tau\ \texttt{"empty"}) \mapsto Opt\ \text{"\_"})\ \hat{\bigcup}\ expr\end{aligned}$$

The first argument *op* of *int_appl* is the root operator of the application. The context argument *c* contains all context information, and the final argument is the list of interior arguments. The language associated with these arguments by *int_appl*, is the interior of the operator *op* in which the argument placeholders have been replaced by interior arguments. An *interior argument* is either an optional placeholder (in case of partial applications) or an expression.

Units are those expressions which don't have an argument on the outside. A unit is either a predefined identifier, or an expression between parentheses or an application of an interior operator:

$$\begin{aligned}& \wedge\ unit = Context \times Expr ::\\ & \quad \overset{<}{\bigcup}(\ c : Context; id : PredefId; e : Expr; op : \{get\_ops\ c\} \cap InteriorOp\ .\\ & \quad\quad (c, (\texttt{"const"}, id)) \mapsto \underline{id}\ \hat{\bigcup}\\ & \quad\quad (c, e) \mapsto \underline{\text{"("}} \diamond expr\ (c, e) \diamond \underline{\text{")"}}\ \hat{\bigcup}\\ & \quad\quad (c, appl\_node\ (op, es)) \mapsto int\_appl\ (op, c, es))\end{aligned}$$

A *default application* is a default application or a unit, followed by a unit:

$$\wedge\ default\_appl = Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; f, x : Expr^2\ .$$
$$(c, (\texttt{"appl"}, f, x)) \mapsto default\_appl'\ (c, f) \diamond unit\ (c, x))$$
$$\wedge default\_appl' = default\_appl\ \hat{\bigcup}\ unit$$

A *prefix application* is an interior application of a prefix operator followed by a prefix argument. A *prefix argument* is either an optional placeholder, or a prefix expression:

$$\wedge\ prefix\_appl = Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; e : Expr; op : \{get\_ops\ c\} \cap PrefixOp\ .$$
$$(c, appl\_node\ (op, es \prec e)) \mapsto int\_appl\ (op, c, es) \diamond prefix\_arg\ (c, e))$$
$$\wedge prefix\_appl' = prefix\_appl\ \hat{\bigcup}\ default\_appl'$$
$$\wedge prefix\_arg = \overset{<}{\bigcup}(c : Context\ .\ (c, \tau\ \texttt{"empty"}) \mapsto Opt\ \texttt{"\_"})\ \hat{\bigcup}\ prefix\_appl'$$

A *postfix application* is a postfix argument followed by an interior application of a postfix operator. A *postfix argument* is either an optional placeholder for partial applications, or a postfix expression:

$$\wedge\ postfix\_appl = Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; e : Expr; op : \{get\_ops\ c\} \cap PostfixOp\ .$$
$$(c, appl\_node\ (op, e \succ es)) \mapsto postfix\_arg\ (c, e) \diamond int\_appl\ (op, c, es))$$
$$\wedge postfix\_appl' = postfix\_appl\ \hat{\bigcup}\ prefix\_appl'$$
$$\wedge postfix\_arg = \overset{<}{\bigcup}(c : Context\ .\ (c, \tau\ \texttt{"empty"}) \mapsto Opt\ \texttt{"\_"})\ \hat{\bigcup}\ postfix\_appl'$$

Infix applications are the most complex form of applications, because they depend on the precedence relations supplied in the context. An infix application is an application of an infix or variadic operator:

$$\wedge\ infix\_appl = Context \times Expr ::$$
$$\overset{<}{\bigcup}(op, c, e : \mathcal{D}\ op\_appl\ .\ (c, e) \mapsto op\_appl\ (op, c, e))$$
$$\wedge infix\_appl' = infix\_appl\ \hat{\bigcup}\ postfix\_appl'$$

An operator application of an infix operator consists of a left argument of that operator followed by an interior of that operator and a right argument of that operator. If the operator is a variadic operator, then the application starts with a left argument of that operator followed by one or more right arguments of that operator separated by the operator symbol of that operator:

$$\wedge\ op\_appl = Operator \times Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; iop : \{get\_ops\ c\} \cap InfixOp; vop : \{get\_ops\ c\} \cap VariadicOp;$$
$$e, e' : Expr^2; es : Expr^* .$$
$$(iop, c, appl\_node\ (iop, e \succ es \prec e')) \mapsto$$
$$left\_arg\ (iop, c, e) \diamond int\_appl\ (iop, c, es) \diamond right\_arg\ (iop, c, e')\ \hat{\bigcup}$$
$$(vop, c, appl\_node\ (vop, e \succ es \prec e')) \mapsto$$
$$left\_arg\ (vop, c, e) \diamond Cat\ (i : \mathcal{D}\ es\ .\ \underline{\alpha\ vop} \diamond right\_arg\ (vop, c, es\ i)) \diamond$$
$$\underline{\alpha\ vop} \diamond right\_arg\ (vop, c, e'))$$

A *left argument* of an operator is either an optional placeholder, or a postfix expression or an operator application of an operator which is allowed as left argument of the first operator by the precedence relations supplied in the context:

$$\wedge\ left\_arg = Operator \times Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; \prec, \succ := get\_rels\ c; e : Expr;$$
$$op, lop : Operator^2 \wedge lop \prec op \wedge \neg(op \in VariadicOp \wedge op = lop) .$$
$$(op, c, \tau\ \texttt{"empty"}) \mapsto Opt\ \texttt{"\_"}\ \hat{\bigcup}$$
$$(op, c, e) \mapsto postfix\_appl'\ (c, e)\ \hat{\bigcup}$$
$$(op, c, e) \mapsto op\_appl\ (lop, c, e))$$

Note that it is not allowed for a variadic operator to have itself as argument, because that would result in ambiguous parsings for variadic applications having more than two arguments.

In the same way, a *right argument* of an operator is either an optional placeholder, or a postfix expression or an operator application of an operator which is allowed as right argument of the first operator:

$$\wedge\ right\_arg = Operator \times Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; \prec, \succ := get\_rels\ c; e : Expr;$$
$$op, rop : Operator^2 \wedge op \succ rop \wedge \neg(op \in VariadicOp \wedge op = rop) .$$
$$(op, c, \tau\ \texttt{"empty"}) \mapsto Opt\ \texttt{"\_"}\ \hat{\bigcup}$$
$$(op, c, e) \mapsto postfix\_appl'\ (c, e)\ \hat{\bigcup}$$
$$(op, c, e) \mapsto op\_appl\ (rop, c, e))$$

A *tuple* is a sequence of two or more infix expressions, separated by commas:

$$\wedge\ tuple = Context \times Expr ::$$
$$\overset{<}{\bigcup}(\ c : Context; t : Expr^* \wedge \#t > 1 .$$
$$(c, (\texttt{"tup"}, t)) \mapsto$$
$$Cat\ (i : \mathcal{D}\ t\ .\ (i = 0\ ?\ Empty \dagger \underline{\texttt{","}})) \diamond infix\_appl'\ (c, t\ i)))$$
$$\wedge tuple' = tuple\ \hat{\bigcup}\ infix\_appl'$$

Funmath has three different notations for function abstraction. All notations contain a *binding* and an expression. The context of the expression is the context returned by the binding after receiving the context of the environment.

$$
\begin{array}{l}
\wedge\ abstraction = Context \times Expr ::\\
\quad \overset{<}{\bigcup}(\ c, c' : Context^2; b : Binding; e : Expr\ .\\
\qquad (c, (\text{"}\texttt{abstr}\text{"}, b, e)) \mapsto binding\ (c, c', b) \diamond \underline{\text{"."}} \diamond abstraction'\ (c', e)\ \hat{\bigcup}\\
\qquad (c, (\text{"}\texttt{vtabstr0}\text{"}, e, b)) \mapsto tuple'\ (c', e) \diamond \underline{\text{"|"}} \diamond binding\ (c, c', b)\ \hat{\bigcup}\\
\qquad (c, (\text{"}\texttt{vtabstr1}\text{"}, (\text{"}\texttt{filter}\text{"}, b, e), vartree\ b)) \mapsto\\
\qquad binding\ (c, c', b) \diamond \underline{\text{"|"}} \diamond abstraction'\ (c', e))\\
\wedge abstraction' = abstraction\ \hat{\bigcup}\ tuple'
\end{array}
$$

The first alternative of *abstraction* describes standard function abstractions which start with a binding followed by a dot and the so called *body* of the abstraction, which is an abstraction expression. The second alternative describes the first form of Van Thienen abstraction, which starts with a tuple expression, the body, followed by a vertical bar and the binding. The meaning of this form is the same as the meaning of the standard form. The final alternative specifies the second Van Thienen abstraction variant. These abstractions start with a binding followed by a vertical bar and an abstraction expression. The meaning of these abstractions is different from the previous forms: the expression is an additional filter to the binding, and the body of the abstraction consists of the nested variable tuple bound by the binding. For instance, `x:Nat;y,z:Nat^2 | x=y+z` means the same as `x:Nat;y,z:Nat^2 with x=y+z . x,(y,z)`.The only difference between the representation of these two abstraction will be the tag of the abstraction node: the first one will be tagged `"vtabstr1"`, while the second one will get the tag `"abstr"`.

Now that all expression forms are defined, we define *expr* as the language function producing all expressions:

$$\wedge expr = abstraction'$$

The following language functions describe the syntax of *bindings*. A *variable unit* is either an existing operator (in which case the placeholders are optional), or a new operator pattern, or a variable tuple between parentheses:

$$
\begin{array}{l}
\wedge\ varunit = Context \times Operator^* \times Binding ::\\
\quad \overset{<}{\bigcup}(\ c : Context; op : \{get\_ops\ c\}; new\_op : Operator \setminus \{get\_ops\ c\};\\
\qquad b : Binding; ops : Operator^*\ .\\
\qquad (c, \varepsilon, (\text{"}\texttt{var}\text{"}, op)) \mapsto Cat\ (i : \mathcal{D}\ op\ .\ op\ i \in Placeholder\ ?\ Opt\ \underline{op\ i} \dagger \underline{op\ i})\ \hat{\bigcup}\\
\qquad (c, \tau\ new\_op, (\text{"}\texttt{var}\text{"}, new\_op)) \mapsto \underline{\text{"("}} \diamond \iota\ new\_op \diamond \underline{\text{")"}}\ \hat{\bigcup}\\
\qquad (c, ops, b) \mapsto \underline{\text{"("}} \diamond vartuple'\ (c, ops, b) \diamond \underline{\text{")"}})
\end{array}
$$

The second component of the argument of *varunit* contains the new operator patterns introduced by the variable unit.

A *variable tuple* is a comma separated list of variable units. Again the second component of the argument contains the new operator patterns introduced by the variable tuple:

$$
\begin{array}{l}
\wedge\ vartuple = Context \times Operator^* \times Binding :: \\
\quad \overset{<}{\bigcup}(\ c : Context; (n : \mathbb{N} \wedge n > 1); opss : (Operator^*)^n; bs : Binding\ . \\
\qquad (c, cat\ opss, (\text{"}\texttt{btup}\text{"}, bs)) \mapsto \\
\qquad Cat\ (i : \square n\ .\ (i = 0\ ?\ Empty \dagger \underline{\text{","}}) \diamond varunit\ (c, opss\ i, bs\ i))) \\
\wedge vartuple' = vartuple \hat{\bigcup} varunit
\end{array}
$$

A *binder* is a variable tuple followed by a colon or becomes symbol and an expression. The new operators introduced in the variable tuple are added to the operator list of the context, provided that the resulting list of operators is valid. If the new operators are invalid then *binder* fails. The representation of the binder itself is added to the local context:

$$
\begin{array}{l}
\wedge\ binder = Context \times Context \times Binding :: \\
\quad \overset{<}{\bigcup}(\ c : Context; ops : Operator^*; b : Binding; e : Expr; \\
\qquad bt := (\text{"}\texttt{type}\text{"}, b, e); ba := (\text{"}\texttt{assign}\text{"}, b, e); \\
\qquad (c' := set\_ops\ (c, get\_ops\ c +\!\!+ ops) \wedge valid\_ops\ (get\_ops\ c')); \\
\qquad ct := set\_loc\ (c', get\_loc\ c \prec bt); ca := set\_loc\ (c', get\_loc\ c \prec ba)\ . \\
\qquad (c, ct, bt) \mapsto vartuple'\ (c, ops, b) \diamond \underline{\text{":"}} \diamond expr\ (c, e) \hat{\bigcup} \\
\qquad (c, ca, ba) \mapsto vartuple'\ (c, ops, b) \diamond \underline{\text{":="}} \diamond expr\ (c, e)) \\
\wedge binder' = binder \hat{\bigcup} (x : \mathcal{D}\ binding\ .\ \underline{\text{"("}} \diamond binding\ x \diamond \underline{\text{")"}})
\end{array}
$$

A *binder tuple* is a semicolon separated list of binders. The context is passed from left to right through the binders:

$$
\begin{array}{l}
\wedge\ bindtuple = Context \times Context \times Binding :: \\
\quad \overset{<}{\bigcup}(\ c, c', c'' : Context^3; bs : Binding^*; b, b' : Binding^2\ . \\
\qquad (c, c'', (\text{"}\texttt{btup}\text{"}, (b, b')) \mapsto binder\ (c, c', b) \diamond \underline{\text{";"}} \diamond binder\ (c', c'', b') \hat{\bigcup} \\
\qquad (c, c'', (\text{"}\texttt{btup}\text{"}, bs \prec b)) \mapsto \\
\qquad bindtuple\ (c, c', (\text{"}\texttt{btup}\text{"}, bs)) \diamond \underline{\text{";"}} \diamond binder\ (c', c'', b) \\
\wedge bindtuple' = bindtuple \hat{\bigcup} binder'
\end{array}
$$

A *binding* is a binder tuple which may be followed by a *filter*, which consists of the symbol `with` and an expression:

$$\wedge\ binding = Context \times Context \times Binding ::$$
$$\overset{<}{\bigcup}(\ c, c' : Context^2; b : Binding; e : Expr; c'' := set\_loc\ (c', get\_loc\ c' \prec e)\ .$$
$$(c, c', b) \mapsto bindtuple'\ (c, c', b)\ \hat{\bigcup}$$
$$(c, c'', (\text{"}\texttt{filter}\text{"}, b, e)) \mapsto bindtuple'\ (c, c', b) \diamond \underline{\text{"}\texttt{with}\text{"}} \diamond expr\ (c', e))$$

This completes the definition of the Funmath expression and binding syntax.

6.7 Declarations

Now that the syntax for operator patterns, precedence patterns, expressions and bindings are complete, we can specify the syntax of the declarations and, finally, the Funmath script syntax.

A *definition* consists of the keyword `def` followed by a binding. The binding is added to the definition list. Note that the local context returned by the binding is erased.

def $definition := Context^2 ::$
$$\overset{<}{\bigcup}(\ c, c' : Context^2; b : Binding\ .$$
$$(c, set\_defs\ (set\_loc\ (c', \varepsilon), get\_defs\ c \prec b)) \mapsto$$
$$\underline{\text{"}\texttt{def}\text{"}} \diamond binding\ (c, c', b))$$

A *specification* starts with the keyword `spec` followed by a binding. The binding is added to the specification list.

def $specification := Context^2 ::$
$$\overset{<}{\bigcup}(\ c, c' : Context^2; b : Binding\ .$$
$$(c, set\_specs\ (set\_loc\ (c', \varepsilon), get\_specs\ c \prec b)) \mapsto$$
$$\underline{\text{"}\texttt{spec}\text{"}} \diamond binding\ (c, c', b))$$

A flaw of this syntax is that it doesn't enforce that only *one* definition is allowed for an identifier, and that the set of defined and specified identifiers must be disjoint. This can be repaired by restricting the type *Context* so that it does not contain such illegal combinations.

A *precedence declaration* is composed of the keyword `par` followed by a precedence pattern. The precedence relations produced by the precedence pattern are added to the precedence relations of the context. The resulting relations are then closed. If the closure is ambiguous, *prec_decl* fails.

def $prec\_decl := Context^2 ::$
$$\overset{<}{\bigcup}(\ c : Context; root : \mathcal{P}\ Operator; rels : (Operator^2 \to \mathbb{B})^2;$$
$$cl := close\ (get\_rels\ c\ \hat{\vee}\ rels) \wedge unamb\_rels\ cl\ .$$
$$(c, set\_rels\ (c, cl)) \mapsto \underline{\text{"}\texttt{par}\text{"}} \diamond prec\_pat\ (root, rels))$$

The language function for *declarations* is now given by:

$$\textbf{def}\ \mathit{declaration} := \mathit{definition}\ \hat{\bigcup}\ \mathit{specification}\ \hat{\bigcup}\ \mathit{prec\_decl}$$

A *Funmath script* consists of declarations:

$$\begin{array}{l}\textbf{def}\ \mathit{funmath} : \mathit{Context} \to \mathcal{L}\ \mathit{Token}\\ \textbf{with}\ \ \mathit{funmath} = \mathit{Context} ::\\ \quad \overset{<}{\bigcup}(\ c, c' : \mathit{Context}^2\ .\\ \qquad (\varepsilon, (0^\bullet \mathit{Operator}^2, 0^\bullet \mathit{Operator}^2), \varepsilon, \varepsilon) \mapsto \mathit{Empty}\ \hat{\bigcup}\\ \qquad c' \mapsto \mathit{funmath}\ \ c \diamond \mathit{declaration}\ (c, c'))\end{array}$$

The empty context is associated with the empty language. The context of a sequence of declarations is obtained by passing the context from left to right through the declarations. This conludes the Funmath declaration syntax.

6.8 Conclusions

We have formally defined the a syntax for the highly context-sensitive Funmath language. The description includes the lexical syntax and the mechanisms for user defined operators and user defined operator precedence. This shows that the methods introduced in Chapter 5 are very general. As a result of the context-sensitivity, some language functions have become complex and hard to read. Therefore, it probably still is desirable to have a separate bare grammar that describes a context-free superset of Funmath by leaving out the context dependencies and the concrete representations.

Chapter 7

Parsing Algorithms for Funmath

7.1 Introduction

The Funmath syntax defined in Chapter 6 provides two very general concepts for user-definable syntax: operator declarations and association declarations. Care has been taken that the declarations don't introduce ambiguities in the language, but we haven't looked yet at the practical realizability of the syntax. This chapter gives the key algorithms needed for parsing Funmath. The time and space complexity of these algorithms is favourable, and make an efficient implementation of the Funmath parser possible.

7.2 Transitive closure of the association relations

The first algorithm we present, deals with the transitive closure of the association relations introduced by association declarations. The algorithm implements the operator *close*, defined in Section 6.5.2. The implementation uses a version of the Warshall algorithm [46], modified to work with our four transitivity rules for the association relations $\prec$ and $\succ$. Below we present the algorithm in Funmath itself.

$$
\begin{array}{l}
\textbf{def}\ \mathit{warshall\_close} : \mathit{Operator}^* \to (\mathit{Operator}^2 \to \mathbb{B})^2 \to (\mathit{Operator}^2 \to \mathbb{B})^2 \\
\textbf{with}\ \ \mathit{warshall\_close} = \\
\quad (\ \mathit{ops} : \mathit{Operator}^*\ . \\
\quad\ (\circ)\langle \mathit{fold}\ \mathcal{F}\rangle(i : \mathcal{D}\ \mathit{ops}\ .\ (\circ)\langle \mathit{fold}\ \mathcal{F}\rangle(j : \mathcal{D}\ \mathit{ops}\ .\ (\circ)\langle \mathit{fold}\ \mathcal{F}\rangle(k : \mathcal{D}\ \mathit{ops}\ . \\
\quad\ [\ m := \mathit{ops}\ i; s := \mathit{ops}\ j; t := \mathit{ops}\ k\ . \\
\quad\quad (\ \prec, \succ : (\mathit{Operator}^2 \to \mathbb{B})^2\ . \\
\quad\quad\quad (\ (\prec) \triangleleft (s,t) \mapsto (s \prec t \vee (s \prec m \vee m \succ s) \wedge m \prec t), \\
\quad\quad\quad\quad (\succ) \triangleleft (s,t) \mapsto (s \succ t \vee s \succ m \wedge (m \succ t \vee t \prec m))) \\
\quad\quad)] \\
\quad))))
\end{array}
$$

Compared to *close*, the implementation *warshall_close* has an additional argument *ops*. This argument is a list of operators that serves as an enumeration of all operators which are in the association relations. If *ops* contains all those operators, then *warshall_close ops* implements *close* correctly. Formally, we can express this property by:

$$\forall(\ ops : Operator^*; \prec, \succ : (Operator^2 \to \mathbb{B})^2\ .\\ \quad \{\!|\prec \hat{\vee} \succ|\!\} \subseteq \{ops\}^2 \Rightarrow warshall\_close\ ops\ (\prec, \succ) = close\ (\prec, \succ))$$

The variables i, j and k of the algorithm range over the indexes of the operators. The local variables m, s and t contain the operators indexed by i, j and k, respectively. The innermost abstraction in the algorithm is the function which performs one update in position (s, t) by taking for both association relations the disjunction of the current value in position (s, t) with the value on the right hand side of $\mapsto$. We assume that the time needed for one update is bounded by a constant.

By using the reduction of function composition, $(\circ)\langle fold\ \mathcal{F}\rangle$, all updates are composed to a chain of $\#ops^3$ updates. This composition computes the closure of a given pair of association relations. The usage of $(\circ)\langle fold\ \mathcal{F}\rangle$ in *warshall_close* corresponds to the use of for-loops in imperative programming languages. This is a consequence of the fact that sequencing of commands in imperative programming languages is a form of function composition, where the commands are functions passing the program state to each other. In *warshall_close* the program state is the pair of relations that is being closed.

The correctness proof of the original Warshall algorithm also holds for our modified version. The complexity is also the same; it runs in $O\ (\#ops^3)$ time, if we assume that one update in the association relations can be done in constant time. If the relations are unambiguous, then they can be used to direct the deterministic parsing of expressions, as will be shown in the next section.

7.3 Parsing infix expressions

This section presents an algorithm which uses the association relations to find the unique parse tree of an infix expression in quadratic time and linear space (measured in the length of the expression). Furthermore, the algorithm will be proved correct.

7.3.1 Simplified infix parse trees

In the presentation of the algorithm and the correctness proof, we abstract from several matters which are not important for infix expression parsing.

- We will only consider infix operators having only *one* operator symbol. We will use the name *Op* for the set of all infix operator symbols, so the operators we consider are given by the language $\underline{\text{"\_"}} \diamond One\ Op \diamond \underline{\text{"\_"}}$.

  spec $Op : \mathcal{T}$ **with** $Op \subseteq OpSymbol$

Parsing more complicated operators will be handled in Section 7.4, which discusses how interiors of operator applications can be parsed.

- We leave out the innermost arguments of the infix operators. In the actual parser, these arguments will end up in the leafs of the parse tree, and therefore don't have effect on the infix operator structure of the parse tree. Below, we represent all innermost arguments by ε nodes.

- Because the association relations don't change during the parsing of expressions, we will treat the relations as global relations on Op satisfying the unambiguity constraint and the transitivity rules:

$$\begin{array}{ll}\textbf{spec} & \prec, \succ : (Op^2 \to \mathbb{B})^2 \\ \textbf{with} & \forall(a, b : Op^2 \,.\, \neg(a \prec b \wedge a \succ b)) \wedge \\ & \quad \forall(\ a, b, c : Op^3 \,. \\ & \qquad (a \prec b \wedge b \prec c \Rightarrow a \prec c) \wedge \\ & \qquad (b \succ a \wedge b \prec c \Rightarrow a \prec c) \wedge \\ & \qquad (a \succ b \wedge c \prec b \Rightarrow a \succ c) \wedge \\ & \qquad (a \succ b \wedge b \succ c \Rightarrow a \succ c))\end{array}$$

Note that $\prec$ and $\succ$ don't have to be opposites, i.e. $a \prec b \equiv b \succ a$ doesn't have to hold.

The following definitions were given in a different form in [39] and will be needed in our correctness proof:
Tree is the type of parse trees of infix operator strings. Because we only consider infix expressions without infix arguments, we can represent the infix arguments by ε nodes. Furthermore, the operator nodes can be represented by a tuple containing the left hand side argument, the operator symbol and the right hand side argument:

$$\textbf{def}\ \mathit{Tree} := \mathit{fix}_{\mathcal{T},\subseteq}\ (\mathit{Tree} : \mathcal{T}\ .\ \iota\ \varepsilon \cup \mathit{Tree} \times \mathit{Op} \times \mathit{Tree})$$

The operator *flat* flattens a parse tree:

$$\begin{array}{ll}\textbf{def} & \mathit{flat} : \mathit{Tree} \to \mathit{Op}^* \\ \textbf{with} & \mathit{flat} = \ (\varepsilon \mapsto \varepsilon) \triangleright \\ & \qquad (l, op, r : \mathit{Tree}\ .\ \mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ \mathit{flat}\ r)\end{array}$$

The predicate *valid* checks if a tree satisfies the constraints implied by the association relations $\prec$ and $\succ$.

$$\begin{array}{ll}\textbf{def} & \mathit{valid} : \mathit{Tree} \to \mathbb{B} \\ \textbf{with} & \mathit{valid} = \ (\varepsilon \mapsto 1) \triangleright \\ & \qquad (l, op, r : \mathit{Tree}\ .\ \mathit{lvalid}\ (l, op) \wedge \mathit{rvalid}\ (op, r))\end{array}$$

$$\begin{array}{l}\textbf{def } \mathit{lvalid} : \mathit{Tree} \times \mathit{Op} \to \mathbb{B} \\ \textbf{with } \mathit{lvalid} = \ (x := \varepsilon; op : \mathit{Op} \ .\ 1) \triangleright \\ \qquad\qquad ((l, op, r) : \mathit{Tree}; op' : \mathit{Op} \ .\ \mathit{valid}\ (l, op, r) \wedge op \prec op')\end{array}$$

$$\begin{array}{l}\textbf{def } \mathit{rvalid} : \mathit{Op} \times \mathit{Tree} \to \mathbb{B} \\ \textbf{with } \mathit{rvalid} = \ (op : \mathit{Op}; x := \varepsilon \ .\ 1) \triangleright \\ \qquad\qquad (op' : \mathit{Op}; (l, op, r) : \mathit{Tree} \ .\ \mathit{valid}\ (l, op, r) \wedge op' \succ op)\end{array}$$

Note that if we define the relation *in* which checks if an operator occurs in a tree by

$$\begin{array}{l}\textbf{def } (— \ \mathit{in} \ —) : \mathit{Op} \times \mathit{Tree} \to \mathbb{B} \\ \textbf{with } (\mathit{in}) = \ (op : \mathit{Op}; x := \varepsilon \ .\ 0) \triangleright \\ \qquad\qquad (op' : \mathit{Op}; (l, op, r) : \mathit{Tree} \ .\ (op'\ \mathit{in}\ l) \vee (op' = op) \vee (op'\ \mathit{in}\ r))\end{array}$$

then the predicates *lvalid* and *rvalid* can also be given by:

$$\begin{array}{l}\mathit{lvalid}\ (op, t) \equiv \mathit{valid}\ t \wedge \forall (op' : \mathit{Op} \ .\ op'\ \mathit{in}\ t \Rightarrow op' \prec op) \\ \mathit{rvalid}\ (op, t) \equiv \mathit{valid}\ t \wedge \forall (op' : \mathit{Op} \ .\ op'\ \mathit{in}\ t \Rightarrow op \succ op')\end{array}$$

These properties provide better understanding of the notion of validness, but the recursive definitions are more suitable for use in inductive proofs.

7.3.2 The algorithm

The parser *buildtree* is based on the tree insert algorithm *ins*. It is the left to right reduction of *ins*. We could also have defined *buildtree* as a right to left reduction of another insert algorithm *ins'*, but as parsing is usually done from left to right this would not be a natural choice and it would needlessly complicate efficient implementation. The algorithm *ins* takes a tree and an operator and tries to insert the operator as deep as possible into the rightmost path of the tree. This is done using the association relations $\prec$ and $\succ$ as follows:

- If the root operator of the given tree is a valid left argument of the given operator then a tree will be returned with the given tree as left subtree, the given operator as operator and the empty tree as right subtree will be returned.
- If the given operator is a valid right argument of the root operator of the given tree then the given operator is inserted into the right subtree of the given tree by a recursive call of *ins*.

$$\begin{array}{l}\textbf{def } \mathit{ins} : \mathit{Tree} \times \mathit{Op} \rightharpoonup \mathit{Tree} \\ \textbf{with } \mathit{ins} = \ (x := \varepsilon; op : \mathit{Op} \ .\ \varepsilon, op, \varepsilon) \triangleright \\ \qquad\qquad (\ (l, op, r) : \mathit{Tree}; op' : \mathit{Op} \ . \\ \qquad\qquad\ op \prec op' \ ?\ ((l, op, r), op', \varepsilon)\ \dagger \\ \qquad\qquad\ op \succ op' \ ?\ [r' := \mathit{ins}\ (r, op') \ .\ r' \neq \bot \ ?\ (l, op, r')])\end{array}$$

def $buildtree : Op^* \rightharpoonup Tree$
with $buildtree = ins \not\rightarrow_{\varepsilon}$

Note that for $t : Tree$ and $op' : Op$ there are two situations in which ins fails, i.e. $ins\ (t, op') = \bot$:

1. $t = (l, op, r) \wedge \neg(op \prec op') \wedge \neg(op \succ op')$
2. $t = (l, op, r) \wedge op \succ op' \wedge ins\ (r, op') = \bot$

All these failures are caused by adjacent operators for which no association was defined. In such case, the user must either place parentheses around one of the operators or supply an association relation between the operators.

7.3.3 Correctness

Definitions

We need two more operators for our correctness proof. The function rm_up replaces the rightmost operator node of the given tree with its left subtree.

def $rm\_up : (Tree \setminus \iota\ \varepsilon) \rightarrow Tree$
with $rm\_up = (l, op, r : Tree \setminus \iota\ \varepsilon\ .\ r = \varepsilon\ ?\ l \dagger l, op, rm\_up\ r)$

The function rm_op yields the rightmost operator of the given tree.

def $rm\_op : (Tree \setminus \iota\ \varepsilon) \rightarrow Op$
with $rm\_op = (l, op, r : Tree \setminus \iota\ \varepsilon\ .\ r = \varepsilon\ ?\ op \dagger rm\_op\ r)$

Lemmas

The following lemma states that rm_up preserves the validity of the given tree.

Lemma 3

$$\forall(t : Tree \setminus \iota\ \varepsilon\ .\ valid\ t \Rightarrow valid\ (rm\_up\ t))$$

Proof

By induction on the length of the rightmost path of t.

- Base, $t = (l, op, \varepsilon)$

$$\begin{array}{lll} valid\ (rm\_up\ t) & & \\ \equiv & \{t = (l, op, \varepsilon)\} & valid\ (rm\_up\ (l, op, \varepsilon)) \\ \equiv & \{\text{def } rm\_up\} & valid\ l \\ \Leftarrow & \{valid\ t \Rightarrow valid\ l\} & valid\ t \end{array}$$

- Induction step, $t = (l, op, r), r = (l', op', r')$
  IH: $valid\ r \Rightarrow valid\ (rm\_up\ r)$

  $valid\ (rm\_up\ t)$

$\equiv$	$\{t = (l, op, r)\}$	$valid\ (rm\_up\ (l, op, r))$
$\equiv$	$\{\neg(r = \varepsilon)$, def $rm\_up\}$	$valid\ (l, op, rm\_up\ .\ r)$
$\equiv$	{def $valid$}	$lvalid\ (l, op) \wedge rvalid\ (op, rm\_up\ r)$
$\equiv$	$\{valid\ t \Rightarrow lvalid\ (l, op)\}$	$rvalid\ (op, rm\_up\ r)$

  Now we can distinguish three cases:

  1. $r' = \varepsilon \wedge l' = \varepsilon$

     $rvalid\ (op, rm\_up\ r)$

$\equiv$	$\{r = (\varepsilon, op, \varepsilon)$, def $rm\_up\}$	$rvalid\ (op, \varepsilon)$
$\equiv$	{def $rvalid$}	1

  2. $r' = \varepsilon \wedge l' = (l'', op'', r'')$

     $rvalid\ (op, rm\_up\ r)$
     $\equiv$ $\{\ r = (l', op', \varepsilon)$, def $rm\_up\ \}$
     $rvalid\ (op, l')$
     $\equiv$ $\{\ l' = (l'', op'', r'')$, def $rvalid\ \}$
     $valid\ l' \wedge op \succ op''$
     $\Leftarrow$ $\{\ valid\ t \Rightarrow valid\ r,\ valid\ r \Rightarrow valid\ l'\ \}$
     $valid\ t \wedge op \succ op''$
     $\Leftarrow$ $\{$ transitivity rule $\succ, \prec\ \}$
     $valid\ t \wedge op \succ op' \wedge op'' \prec op'$
     $\Leftarrow$ $\{\ valid\ t \Rightarrow op \succ op'$,$valid\ r \Rightarrow op'' \prec op'\ \}$
     $valid\ t$

  3. $\neg(r' = \varepsilon)$

     $rvalid\ (op, rm\_up\ r)$
     $\equiv$ $\{\ r = (l', op', r')$, $\neg(r' = \varepsilon)$, def $rm\_up\ \}$
     $rvalid\ (op, (l', op', rm\_up\ r'))$
     $\equiv$ $\{$ def $rvalid\ \}$
     $valid\ (l', op', rm\_up\ r') \wedge op \succ op'$
     $\Leftarrow$ $\{\ valid\ t \Rightarrow rvalid\ (op, r)$, $rvalid\ (op, r) \Rightarrow op \succ op'\ \}$
     $valid\ (l', op', rm\_up\ .\ r') \wedge valid\ t$
     $\equiv$ $\{\ r = (l', op', r')$, def $rm\_up\ \}$

$$
\begin{array}{ll}
 & \mathit{valid}\ (\mathit{rm\_up}\ r) \wedge \mathit{valid}\ t \\
\Leftarrow & \qquad \{\, \mathit{valid}\ t \Rightarrow \mathit{valid}\ r,\ \text{IH} \,\} \\
 & \mathit{valid}\ t
\end{array}
$$

So in all cases we have $\mathit{valid}\ t \Rightarrow \mathit{valid}\ (\mathit{rm\_up}\ t)$

□

The next lemma shows that the flattening of a tree t equals the flattening of $\mathit{rm\_up}\ t$ followed by the rightmost operator of t.

Lemma 4

$$\forall (t : \mathit{Tree} \setminus \iota\ \varepsilon\ .\ \mathit{flat}\ t = \mathit{flat}\ (\mathit{rm\_up}\ t) \prec \mathit{rm\_op}\ t)$$

Proof
Again, by induction on the length of the rightmost path of t.

- Base, $t = (l, op, \varepsilon)$

$\mathit{flat}\ (\mathit{rm\_up}\ t) \prec \mathit{rm\_op}\ t$

$$
\begin{array}{lll}
= & \{t = (l, op, \varepsilon),\ \text{def}\ \mathit{rm\_up}, \mathit{rm\_op}\ \} & \mathit{flat}\ l \prec op \\
= & \{\text{def} \prec\} & \mathit{flat}\ l +\!\!+ \tau\ op \\
= & \{\varepsilon = \mathit{unit}\ \mathcal{U}^{\omega}(+\!\!+)\} & \mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ \varepsilon \\
= & \{\text{def}\ \mathit{flat}\} & \mathit{flat}\ (l, op, \varepsilon) \\
= & \{t = (l, op, \varepsilon)\} & \mathit{flat}\ t
\end{array}
$$

- Induction step, $t = (l, op, r),\ r = (l', op', r')$
  IH: $\mathit{flat}\ r = \mathit{flat}\ (\mathit{rm\_up}\ r) \prec \mathit{rm\_op}\ r$

$$
\begin{array}{ll}
 & \mathit{flat}\ (\mathit{rm\_up}\ t) \prec \mathit{rm\_op}\ t \\
= & \qquad \{\, t = (l, op, r), \neg(r = \varepsilon),\ \text{def}\ \mathit{rm\_up}, \mathit{rm\_op}\ \} \\
 & \mathit{flat}\ (l, op, \mathit{rm\_up}\ r) \prec \mathit{rm\_op}\ r \\
= & \qquad \{\, \text{def}\ \mathit{flat}\ \} \\
 & (\mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ \mathit{flat}\ (\mathit{rm\_up}\ r)) \prec \mathit{rm\_op}\ r \\
= & \qquad \{\, \text{def} \prec \,\} \\
 & (\mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ \mathit{flat}\ (\mathit{rm\_up}\ r)) +\!\!+ \tau\ \mathit{rm\_op}\ r \\
= & \qquad \{\, \text{associativity of} +\!\!+ \,\} \\
 & \mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ (\mathit{flat}\ (\mathit{rm\_up}\ r) +\!\!+ \tau\ \mathit{rm\_op}\ r) \\
= & \qquad \{\, \text{def} \prec \,\} \\
 & \mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ (\mathit{flat}\ (\mathit{rm\_up}\ r) \prec \tau\ \mathit{rm\_op}\ r) \\
= & \qquad \{\, \text{IH} \,\}
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{flat}\ l +\!\!+ \tau\ op +\!\!+ \mathit{flat}\ r \\
= & \qquad \{\, t = (l, op, r),\ \text{def}\ \mathit{flat}\,\} \\
& \mathit{flat}\ t
\end{array}
$$

□

The next lemma states that the rightmost operator of a valid tree, whose right subtree is not empty, is a valid right argument of the root operator of the tree.

Lemma 5

$$\forall(op : Op; r : Tree \setminus \iota\ \varepsilon\ .\ \mathit{rvalid}\ (op, r) \Rightarrow op \succ \mathit{rm\_op}\ r)$$

Proof
Using induction on the length of the rightmost path of r

- Base, $r = (l, op', \varepsilon)$

$$
\begin{array}{lll}
op \succ \mathit{rm\_op}\ r & & \\
\equiv & \{r = (l, op', \varepsilon),\ \text{def}\ \mathit{rm\_op}\} & op \succ op' \\
\Leftarrow & \{\text{def}\ \mathit{rvalid}\} & \mathit{rvalid}\ (op, (l, op', \varepsilon)) \\
\equiv & \{r = (l, op', \varepsilon)\} & \mathit{rvalid}\ (op, r)
\end{array}
$$

- Induction step, $r = (l, op', r'), \neg(r' = \varepsilon)$
  IH: $\mathit{rvalid}\ (op', r') \Rightarrow op' \succ \mathit{rm\_op}\ r'$

$$
\begin{array}{lll}
op \succ \mathit{rm\_op}\ r & & \\
\equiv & \{r = (l, op', r'),\ \neg(r' = \varepsilon),\ \text{def}\ \mathit{rm\_op}\} & op \succ \mathit{rm\_op}\ r' \\
\Leftarrow & \{\text{transitivity rule}\ \succ, \succ\} & op \succ op' \wedge op' \succ \mathit{rm\_op}\ r' \\
\Leftarrow & \{\mathit{rvalid}\ (op, r) \Rightarrow \mathit{rvalid}\ (op', r'), \text{IH}\} & op \succ op' \wedge \mathit{rvalid}\ (op, r) \\
\Leftarrow & \{\mathit{rvalid}\ (op, r) \Rightarrow op \succ op'\} & \mathit{rvalid}\ (op, r)
\end{array}
$$

□

The following lemma is the most important one; it proves that if we insert the rightmost operator of a valid tree t in $\mathit{rm\_up}\ t$ we obtain the original tree t.

Lemma 6

$$\forall(t : Tree \wedge \mathit{valid}\ t\ .\ \mathit{ins}\ (\mathit{rm\_up}\ t, \mathit{rm\_op}\ t) = t)$$

Proof
Using induction on the length of the rightmost path of t

- Base, $t = (l, op, \varepsilon)$

$$
\begin{array}{ll}
 & ins\ (rm\_up\ t, rm\_op\ t) \\
= & \quad \{\ t = (l, op, \varepsilon),\ \text{def}\ rm\_op, rm\_up\ \} \\
 & ins\ (l, op) \\
= & \quad \{\ valid\ t \Rightarrow lvalid\ (l, op),\ lvalid\ (l, op) \Rightarrow l = \varepsilon \vee l\ 1 \prec op,\ \text{def}\ ins\ \} \\
 & (l, op, \varepsilon) \\
= & \quad \{\ t = (l, op, \varepsilon)\ \} \\
 & t
\end{array}
$$

- Induction step, $t = (l, op, r)$, $\neg(r = \varepsilon)$
  IH: $valid\ r \Rightarrow ins\ (rm\_up\ r, rm\_op\ r) = r$

$$
\begin{array}{ll}
 & ins\ (rm\_up\ t, rm\_op\ t) = t \\
\equiv & \quad \{\ t = (l, op, r), \neg(r = \varepsilon),\ \text{def}\ rm\_up, rm\_op\ \} \\
 & ins\ ((l, op, rm\_up\ r), rm\_op\ r) = (l, op, r) \\
\Leftarrow & \quad \{\ \text{def}\ ins\ \} \\
 & op \succ rm\_op\ r \wedge ins\ (rm\_up\ r, rm\_op\ r) = r \\
\Leftarrow & \quad \{\ valid\ t \Rightarrow rvalid\ (op, r),\ \text{Lemma 5}\ \} \\
 & valid\ t \wedge ins\ (rm\_up\ r, rm\_op\ r) = r \\
\Leftarrow & \quad \{\ valid\ t \Rightarrow valid\ r, \text{IH}\ \} \\
 & valid\ t
\end{array}
$$

  So we have $valid\ t \Rightarrow ins\ (rm\_up\ t, rm\_op\ t) = t$

□

The following lemma states that *ins* is validity preserving.

Lemma 7

$$\forall(t : Tree; op : Op\ .\ ins\ (t, op) \in Tree \wedge valid\ t \Rightarrow valid\ (ins\ (t, op)))$$

Proof
By induction on the length of the rightmost path of t

- Base, $t = \varepsilon$

$$
\begin{array}{lll}
valid\ (ins\ (t, op)) & & \\
\equiv & \{t = \varepsilon, \text{def}\ ins\} & valid\ (\varepsilon, op, \varepsilon) \\
\equiv & \{\text{def}\ valid\} & 1
\end{array}
$$

- Induction step, $t = (l, op', r)$
  IH: $ins\ (r, op) \in Tree \wedge valid\ r \Rightarrow valid\ (ins\ (r, op))$
  Because $ins\ (t, op) \in Tree \Rightarrow op' \prec op \vee op' \succ op$ we can distinguish the following two cases:

  1. $op' \prec op$

     $valid\ (ins\ (t, op))$

$\equiv$	$\{t = (l, op', r),\ op' \prec op,$ def $ins\}$	$valid\ (t, op', \varepsilon)$
$\equiv$	{def $valid$}	$lvalid\ (t, op') \wedge rvalid\ (op', \varepsilon)$
$\equiv$	{def $rvalid$}	$lvalid\ (t, op')$
$\equiv$	$\{t = (l, op', r),$ def $lvalid\}$	$valid\ t \wedge op' \prec op$
$\equiv$	$\{op' \prec op\}$	$valid\ t$

  2. $op' \succ op$

     $$
     \begin{array}{ll}
     & valid\ (ins\ (t, op)) \\
     \equiv & \quad \{\ t = (l, op', r),\ op' \succ op,\ ins\ (t, op) \in Tree,\ \text{def}\ ins\ \} \\
     & valid\ (l, op', ins\ (r, op)) \\
     \equiv & \quad \{\ \text{def}\ valid\ \} \\
     & lvalid\ (l, op') \wedge rvalid\ (op', ins\ (r, op)) \\
     \Leftarrow & \quad \{\ valid\ t \Rightarrow lvalid\ (l, op')\ \} \\
     & rvalid\ (op', ins\ (r, op))
     \end{array}
     $$

     Because $ins\ (r, op) \in Tree$ we only have to consider these three cases:

     (a) $r = \varepsilon$

$rvalid\ (op', ins\ (r, op))$		
$\equiv$	$\{r = \varepsilon,$ def $ins\}$	$rvalid\ (op', (\varepsilon, op, \varepsilon))$
$\equiv$	{def $rvalid$}	$valid\ (\varepsilon, op, \varepsilon) \wedge op' \succ op$
$\equiv$	$\{op' \succ op\}$	$valid\ (\varepsilon, op, \varepsilon)$
$\equiv$	{def $valid$}	1

     (b) $r = (l', op'', r') \wedge op'' \prec op$

     $$
     \begin{array}{ll}
     & rvalid\ (op', ins\ (r, op)) \\
     \equiv & \quad \{\ r = (l', op'', r'),\ op'' \prec op,\ \text{def}\ ins\ \} \\
     & rvalid\ (op', (r, op, \varepsilon)) \\
     \equiv & \quad \{\ \text{def}\ rvalid\ \} \\
     & valid\ (r, op, \varepsilon) \wedge op' \succ op \\
     \equiv & \quad \{\ op' \succ op\ \} \\
     & valid\ (r, op, \varepsilon)
     \end{array}
     $$

$$
\begin{array}{ll}
\equiv & \{\,\text{def } valid\,\} \\
& lvalid\ (r, op) \wedge rvalid\ (op, \varepsilon) \\
\equiv & \{\,\text{def } rvalid\,\} \\
& lvalid\ (r, op) \\
\equiv & \{\,r = (l', op'', r'),\ \text{def } lvalid\,\} \\
& valid\ r \wedge op'' \prec op \\
\equiv & \{\,op'' \prec op\,\} \\
& valid\ r \\
\equiv & \{\,valid\ t \Rightarrow valid\ r\,\} \\
& 1
\end{array}
$$

(c) $r = (l', op'', r') \wedge op'' \succ op$

$$
\begin{array}{ll}
& rvalid\ (op', ins\ (r, op)) \\
\equiv & \{\,r = (l', op'', r'),\ op'' \succ op,\ ins\ (r, op) \in Tree,\ \text{def } ins\,\} \\
& rvalid\ (op', (l', op'', ins\ (r', op))) \\
\equiv & \{\,\text{def } rvalid\,\} \\
& valid\ (l', op'', ins\ (r', op)) \wedge op' \succ op'' \\
\equiv & \{\,valid\ t \Rightarrow rvalid\ (op', r),\ rvalid\ (op', r) \Rightarrow op' \succ op''\,\} \\
& valid\ (l', op'', ins\ (r', op)) \\
\equiv & \{\,r = (l', op'', r'),\ op'' \succ op,\ \text{def } ins\,\} \\
& valid\ (ins\ (r, op)) \\
\equiv & \{\,op' \succ op \wedge ins\ (t, op) \in Tree \Rightarrow ins\ (r, op) \in Tree, \text{IH}\,\} \\
& 1
\end{array}
$$

So in all cases we have $ins\ (r, op) \in Tree \wedge valid\ r \Rightarrow valid\ (ins\ (r, op))$.

□

The final lemma shows that $ins\ (t, op)$ is a tree of $flat\ t \prec op$.

Lemma 8

$$\forall (t : Tree; op : Op \wedge ins\ (t, op) \in Tree\ .\ flat\ (ins\ (t, op)) = flat\ t \prec op)$$

Proof
By induction on the length of the rightmost path of t

- Base, $t = \varepsilon$

$flat\ (ins\ (t, op))$

$$
\begin{array}{lll}
= & \{t = \varepsilon, \text{ def } ins\} & flat\ (\varepsilon, op, \varepsilon) \\
= & \{\text{def } flat\} & \varepsilon +\!\!+_\tau\ op +\!\!+ \varepsilon \\
= & \{\varepsilon = unit\ \mathcal{U}^\omega(+\!\!+)\} & \varepsilon +\!\!+_\tau\ op \\
= & \{\text{def} \prec\} & \varepsilon \prec op \\
= & \{t = \varepsilon, \text{ def } flat\} & flat\ t \prec op
\end{array}
$$

- Induction step, $t = (l, op', r)$
  IH: $ins\ (r, op) \in Tree \Rightarrow flat\ (ins\ (r, op)) = flat\ r \prec op$
  Because we have $ins\ (t, op) \in Tree \Rightarrow op' \prec op \vee op' \succ op$ the following two cases have to be distinguished:

  1. $op' \prec op$

     $flat\ (ins\ (t, op))$

     $$
     \begin{array}{lll}
     = & \{t = (l, op', r),\ op' \prec op, \text{ def } ins\} & flat\ (t, op, \varepsilon) \\
     = & \{\text{def } flat\} & flat\ t +\!\!+_\tau\ op +\!\!+ \varepsilon \\
     = & \{\varepsilon = unit\ \mathcal{U}^\omega(+\!\!+)\} & flat\ t +\!\!+_\tau\ op \\
     = & \{\text{def} \prec\} & flat\ t \prec op
     \end{array}
     $$

  2. $op' \succ op$

     $$
     \begin{array}{ll}
     & flat\ (ins\ (t, op)) \\
     = & \quad \{\ t = (l, op', r),\ op' \succ op,\ ins\ (t, op) \in Tree, \text{ def } ins\ \} \\
     & flat\ (l, op', ins\ (r, op)) \\
     = & \quad \{\ \text{def } flat\ \} \\
     & flat\ l +\!\!+_\tau\ op' +\!\!+ flat\ (ins\ (r, op)) \\
     = & \quad \{\ op' \succ op \wedge ins\ (t, op) \in Tree \Rightarrow ins\ (r, op) \in Tree, \text{ IH}\ \} \\
     & flat\ l +\!\!+_\tau\ op' +\!\!+ (flat\ r \prec op) \\
     = & \quad \{\ \text{def} \prec\ \} \\
     & flat\ l +\!\!+_\tau\ op' +\!\!+ (flat\ r +\!\!+_\tau\ op) \\
     = & \quad \{\ \text{associativity of } +\!\!+\ \} \\
     & (flat\ l +\!\!+_\tau\ op' +\!\!+ flat\ r) +\!\!+_\tau\ op \\
     = & \quad \{\ \text{def} \prec\ \} \\
     & (flat\ l +\!\!+_\tau\ op' +\!\!+ flat\ r) \prec op \\
     = & \quad \{\ t = (l, op', r), \text{ def } flat\ \} \\
     & flat\ t \prec op
     \end{array}
     $$

□

The correctness proof

The algorithm *buildtree* is correct if it yields the valid parse tree of the given infix operator string if it exists, and otherwise fails. In formal terms:

$$\forall(s : Op^*; t : Tree \;.\; buildtree\ s = t \equiv valid\ t \wedge flat\ t = s)$$

First we will prove that *buildtree* yields the valid parse tree t of the given string $s = flat\ t$ if it exists.

Theorem 9

$$\forall(s : Op^*; t : Tree \;.\; valid\ t \wedge flat\ t = s \Rightarrow buildtree\ s = t)$$

which is equivalent to:

$$\forall(t : Tree \wedge valid\ t \;.\; buildtree\ (flat\ t) = t)$$

Proof
By induction on the length of $flat\ t$

- Base, $t = \varepsilon$

  $buildtree\ (flat\ t)$

=	$\{t = \varepsilon$, def $flat\}$	$buildtree\ \varepsilon$
=	{def $buildtree$}	$ins \not\rightarrow_\varepsilon \varepsilon$
=	{def $\not\rightarrow$}	ε
=	$\{t = \varepsilon\}$	t

- Induction step, $\neg(t = \varepsilon)$
  IH: $valid\ (rm\_up\ t) \Rightarrow buildtree\ (flat\ (rm\_up\ t)) = rm\_up\ t$

  $buildtree\ (flat\ t)$

=	{def $buildtree$}	$ins \not\rightarrow_\varepsilon (flat\ t)$
=	{Lemma 4, def $\not\rightarrow$}	$ins\ (ins \not\rightarrow_\varepsilon flat\ (rm\_up\ t), rm\_op\ t)$
=	{def $buildtree$}	$ins\ (buildtree\ (flat\ (rm\_up\ t)), rm\_op\ t)$
=	{Lemma 3, IH}	$ins\ (rm\_up\ t, rm\_op\ t)$
=	{Lemma 6}	t

□

Finally, we have to prove the converse of the previous theorem.

Theorem 10

$$\forall(s : Op^*; t : Tree \;.\; buildtree\ s = t \Rightarrow valid\ t \wedge flat\ t = s)$$

which is equivalent to

$$\forall(s : Op^* \wedge buildtree\ s \in Tree\ .\ valid\ (buildtree\ s)) \wedge$$
$$\forall(s : Op^* \wedge buildtree\ s \in Tree\ .\ flat\ (buildtree\ s) = s)$$

These two propositions will be proved separately:

- $\forall(s : Op^* \wedge buildtree\ s \in Tree\ .\ valid\ (buildtree\ s))$

  Proof
  By induction on the length of s

  - Base, $s = \varepsilon$

    $$\begin{array}{lll} valid\ (buildtree\ s) & & \\ \equiv & \{s = \varepsilon, \text{def } buildtree\} & valid\ (ins \not\rightarrow_\varepsilon \varepsilon) \\ \equiv & \{\text{def} \not\rightarrow\} & valid\ \varepsilon \\ \equiv & \{\text{def } valid\} & 1 \end{array}$$

  - Induction step, $s = s' \prec op$
    IH: $buildtree\ s' \in Tree \Rightarrow valid\ (buildtree\ s')$

    $$\begin{array}{ll} & valid\ (buildtree\ s) \\ \equiv & \quad \{\text{def } buildtree\} \\ & valid\ (ins \not\rightarrow_\varepsilon s) \\ \equiv & \quad \{s = s' \prec op, \text{def} \not\rightarrow\} \\ & valid\ (ins\ (ins \not\rightarrow_\varepsilon s', op)) \\ \equiv & \quad \{\text{def } buildtree\} \\ & valid\ (ins\ (buildtree\ s', op)) \\ \Leftarrow & \quad \{buildtree\ s \in Tree \Rightarrow buildtree\ s' \in Tree, \text{Lemma 7}\} \\ & valid\ (buildtree\ s') \\ \equiv & \quad \{buildtree\ s \in Tree \Rightarrow buildtree\ s' \in Tree, \text{IH}\} \\ & 1 \end{array}$$

- $\forall(s : Op^* \wedge buildtree\ s \in Tree\ .\ flat\ (buildtree\ s) = s)$

  Proof
  By induction on the length of s

  - Base, $s = \varepsilon$

    $$\begin{array}{lll} flat\ (buildtree\ s) & & \\ = & \{s = \varepsilon, \text{def } buildtree\} & flat\ (ins \not\rightarrow_\varepsilon \varepsilon) \\ = & \{\text{def} \not\rightarrow\} & flat\ \varepsilon \\ = & \{\text{def } flat\} & \varepsilon \\ = & \{s = \varepsilon\} & s \end{array}$$

– Induction step, $s = s' \prec op$
IH: $buildtree\ s' \in Tree \Rightarrow flat\ (buildtree\ s') = s'$

$$
\begin{array}{ll}
 & flat\ (buildtree\ s) \\
= & \quad \{ \text{def } buildtree \} \\
 & flat\ (ins \not\rightarrow_\varepsilon s) \\
= & \quad \{ s = s' \prec op, \text{def } \not\rightarrow \} \\
 & flat\ (ins\ (ins \not\rightarrow_\varepsilon s', op)) \\
= & \quad \{ \text{def } buildtree \} \\
 & flat\ (ins\ (buildtree\ s', op)) \\
= & \quad \{ buildtree\ s \in Tree \Rightarrow buildtree\ s' \in Tree, \text{Lemma 8} \} \\
 & flat\ (buildtree\ s') \prec op \\
= & \quad \{ buildtree\ s \in Tree \Rightarrow buildtree\ s' \in Tree, \text{IH} \} \\
 & s' \prec op \\
= & \quad \{ s = s' \prec op \} \\
 & s
\end{array}
$$

□

7.3.4 Parse tree uniqueness

The parse tree uniqueness theorem from [39] was not necessary for the correctness proof. Therefore parse tree uniqueness can also be derived from Theorem 9:

$$
\begin{array}{lll}
t & & \\
= & \{valid\ t, \text{Theorem 9}\} & buildtree\ (flat\ t) \\
= & \{flat\ t = flat\ t'\} & buildtree\ (flat\ t') \\
= & \{valid\ t', \text{Theorem 9}\} & t'
\end{array}
$$

So we have $valid\ t \wedge valid\ t' \wedge flat\ t = flat\ t' \Rightarrow t = t'$, which means that there is at most one valid parse tree for any infix operator string.

7.3.5 Complexity of the algorithm

If a string of n operators is passed to *buildtree*, it will do n insert operations, each with as tree argument the result of the previous *ins*. The maximum time needed for an *ins* operation is linear in the length of the rightmost path of the given tree. After m insertions by *buildtree* this length is less than or equal to m. So the total time needed by *buildtree* is bounded by $\sum(m : \Box n\ .\ C \cdot m + D)$ for some real positive C and D. And this summation clearly yields complexity $O\ (n^2)$.

The space needed by *buildtree* has complexity $O(n)$ because each insertion results in the creation of one new node.

7.4 Parsing interiors of operator applications

Section 7.3 showed that the unambiguity and transitivity constraints on the association relations are not only needed for preventing ambiguities, but even make it possible to parse infix expressions efficiently. The unique separation class restriction on operator patterns described in Section 6.4 has similar consequences on parsing efficiency. This restriction makes it possible to divide the set of used operator symbols into four groups, based on whether the operator symbol is an opening and/or closing symbol, so that the first operator symbol of each operator pattern is an opening symbol, the last operator symbol of each pattern is a closing symbol, and all remaining symbols are neither. This classification makes parsing interiors of operator applications easy, because when the parser encounters a symbol that is no opening symbol, this symbol *must* belong to the last opening symbol that doesn't have a matching closing symbol yet. If the symbol encountered is a closing symbol then it completes the interior of the operator application. The only thing left to do then, is to check if the sequence of operator symbols indeed forms an existing operator pattern, and that the interior arguments appear in the places indicated by the operator pattern.

7.5 Implementation

The algorithms described in the previous sections have actually been used in an implementation of a limited version of the Funmath syntax (see [9]). The main limitation of the implemented version is that operator patterns can't be introduced in bindings, so that all operator patterns must be introduced in separate operator declarations. This reduces the context-sensitivity of the original language so much that it is possible to parse the language using *only* the algorithms given in the previous sections. This is done by treating all syntactic constructs for bindings and expressions as regular operators with predefined operator patterns and precedence. After parsing, the application nodes of these operators are converted to abstract syntax nodes for the corresponding syntactical construct. Extending the given algorithms to deal with the extra context-sensitivity of the unrestricted syntax is not hard as long as the bindings precede the expressions in which the patterns introduced in the bindings may be used. In that case, the patterns read in the binding can just be added to the global operator pattern list, and later removed upon exit of the scope of the binding. However, there is one Van Thienen abstraction variant in which the order of binding and body is reversed. An example hereof is the abstraction $n \cdot n \mid n : \mathbb{N}$. These abstractions cannot just be parsed from left to right, but require scanning the binding first and then parsing the body.

7.6 Conclusions

Even though the Funmath language is very flexible and highly context-sensitive, we have presented simple algorithms with which the language can be parsed efficiently. This efficiency could be reached as a result of the restrictions imposed on the user defined operator patterns and operator precedence declarations. Originally, these restrictions only were intended to make the language unambiguous, but their simplicity also made efficient implementation possible. It probably is possible to make the Funmath language even more flexible by finding restrictions which also prevent ambiguity but are less restrictive than the current solution. Preventing ambiguity is not the only requirement, though. A more important requirement is that the user must be able to understand the restrictions. This requirement makes many state of the art parsing techniques unsuitable as base for the Funmath operator pattern and operator precedence mechanisms. Besides, the current solution is much more general than the operator mechanisms of existing languages, which usually only allow user defined infix operators whose precedence is defined by priority numbers.

In this chapter we also have given an example of how the transformational proof style can be used to show the correctness of algorithms.

Appendix A

Precedence of predefined operators

A.1 Logical operators

par $(— \Rightarrow —) \equiv (— \Rightarrow —)$
par $— \Rightarrow (— \Rightarrow —)$
par $(— \Leftarrow —) \equiv (— \Leftarrow —)$
par $(— \Leftarrow —) \Leftarrow —$
par $(— \vee —) \Rightarrow (— \vee —)$
par $(— \vee —) \Leftarrow (— \vee —)$
par $(— \vee —) \vee —$
par $(— \wedge —) \vee (— \wedge —)$
par $(— \wedge —) \wedge —$
par $(— = —) \wedge (— = —)$

A.2 Arithmetic operators

par $(— + —) < (— + —)$
par $(— + —) \leq (— + —)$
par $(— < —) \wedge (— < —)$
par $(— \leq —) \wedge (— \leq —)$

par $(— + —) = (— + —)$
par $((— - —) + —) - —$
par $(— \cdot —) + (— \cdot —)$
par $(— \cdot —) - (— \cdot —)$
par $((— \cdot —)/—) \cdot —$

A.3 Conditional operators

par (— ? —) ǂ (— ? —)
par (— ǂ —) ǂ —
par (— ≡ —) ? (— ≡ —)
par (— ≡ —) ǂ (— ≡ —)

A.4 Set operators

par (— ∈ —) ∧ (— ∈ —)
par (— ∉ —) ∧ (— ∉ —)
par (— ⊆ —) ∧ (— ⊆ —)
par (— = —) ∈ (— = —)
par (— = —) ∉ (— = —)
par (— = —) ⊆ (— = —)

par (— ∪ —) = (— ∪ —)
par (— ∩ —) ∪ (— ∩ —)
par (— \ —) ∩ (— \ —)
par (— \ —) \ —

Bibliography

[1] H. Alblas, *Introduction to Attribute Grammars*, in: H. Alblas and B. Melichar, eds., *International Summer School on Attribute Grammars, Applications and Systems*, Lecture Notes in Computer Science, Vol. 545, pp.1-15, Springer-Verlag, 1991

[2] R. Alur and D.L. Dill, *A theory of timed automata*, Theoretical Computer Science, Vol. 126, pp.183-235, 1994

[3] S. van Bakel, *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*, Ph.D. thesis, Dept. of Computing Science, University of Nijmegen, 1993

[4] H.P. Barendregt, *The lambda calculus; its syntax and semantics*, revised edition, Studies in Logic and the Foundations of Mathematics, North Holland Publishing, 1984.

[5] H. Barendregt, M. Coppo and M. Dezani-Ciancaglini, *A filter lambda model and the completeness of type assignment*, Symbolic Logic, Vol. 48, No. 4, pp.931-940, 1983

[6] J.A. Bergstra, I. Bethke, P.H. Rodenburg, *A Propositional Logic with 4 Values: True, False, Divergent and Meaningless*, Report P9406, Programming Research Group, University of Amsterdam, March 1994

[7] F.A.M. van den Beuken, *Specification and Implementation of a Transformation System for Funmath*, Master's thesis, Faculty of Mathematics and Computing Science, University of Nijmegen, May 1992

[8] F.A.M. van den Beuken, *A deterministic parser for infix expressions using association relations*, in *Documentatie bij Funmath Parser*, Technical Note CSI-N9401, Computing Science Institute, University of Nijmegen, December 1994

[9] F.A.M. van den Beuken, *A deterministic parser for Funmath*, in *Documentatie bij Funmath Parser*, Technical Note CSI-N9401, Computing Science Institute, University of Nijmegen, December 1994

[10] R.T. Boute, *System Semantics and Formal Circuit Description*, IEEE Transactions on Circuits and Systems, Vol.CAS-33, no.12, pp.1219-1231, December 1986

[11] R.T. Boute, *System Semantics: Principles, Application and Implementation*, ACM Transactions on Programming Languages and Systems, Vol.10(1), pp.118-155, January 1988

[12] R.T. Boute, *Conditionals, guards and inverses in Funmath and Comma*, Technical Note, Faculty of Mathematics and Computing Science, University of Nijmegen, November 1990

[13] R.T. Boute, *ESPIRIT Project 881 — FORFUN, Formal description of digital and analog systems by means of functional languages*, ESPIRIT '90 Proceedings of the Anual ESPIRIT Conference, pp.212-226, Kluwer Academic Publishers, Dordrecht, November 1990

[14] R.T. Boute, *A Declarative Formalism Supporting Hardware/Software Co-design*, in: *IFIP International Workshop on Hardware/Software Co-design*, 1992

[15] R.T. Boute, *Fundamentals of Hardware Description Languages and Declarative Languages*, in: Jean P. Mermet ed., *Fundamentals and Standards in Hardware*, pp.3-38, NATO ASI series, Kluwer Academic Publisher, Dordrecht, the Netherlands, 1993

[16] R.T. Boute, *Funmath Illustrated: A declarative formalism and application examples*, Declarative Systems Series No.1, Computing Science Institute, University of Nijmegen, July 1993

[17] R.T. Boute, *Basiswiskunde voor Computerwetenschappen*, lecture notes, INTEC, University of Gent, 1995

[18] M.G.J. van den Brand, *Pregmatic, A Generator for Incremental Programming Environments*, Ph.D. thesis, University of Nijmegen, 1992

[19] Luca Cardelli and Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, Vol.17(4), December 1985

[20] Giuseppe Castagna, *Covariance and contravariance: conflict without a cause*, ACM Transactions on Programming Languages and Systems, Vol.17(3), pp.431-447, 1995

[21] Giuseppe Castagna, Giorgio Ghelli and Giuseppe Longo, *A Calculus for Overloaded Functions with Subtyping*, Information and Computation, Vol.117(1), pp.115-135, February 1995. A preliminary version appeared in ACM Conference on Lisp and Functional Programming, June 1992

[22] M. van Eekelen and R. Plasmeijer, *Concurrent Clean Language Report*, draft version available on WWW from `http://www.cs.kun.nl/ clean/Clean.Cleanbook.html`, 1996

[23] T.E. Forster, *Set theory with a universal set*, Clarendon Press, Oxford, 1992

[24] D. Gries, *Compiler Construction for Digital Computers*, 1971

[25] K. Hanna, N. Daeche, *Implementation of the Veritas design logic*, in: V. Stravidou, T. Melham, R. Boute, eds., *Theorem Provers in Circuit Design*, pp.77-84, North Holland Publishing, Amsterdam, 1992

[26] P. Hudak et al., eds., *Report on the programming language Haskell*, Version 1.2, ACM SIGPLAN Notices 27, 5, May, 1992

[27] J. Huinink, *Towards Comma*, Master's thesis, Faculty of Mathematics and Computing Science, University of Nijmegen, 1996

[28] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1987, IEEE, New York, 1988

[29] P. Jackson, *Nuprl and its use in circuit design*, in: V. Stravidou, T. Melham, R. Boute, eds., *Theorem Provers in Circuit Design*, pp.311-336, North Holland Publishing, Amsterdam, 1992

[30] S.C. Kleene, *On a notation for ordinal numbers*, Journal of Symbolic Logic, Vol.3, pp.150-155, 1938

[31] D.E. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory, Vol.2, pp.127-145, February 1968

[32] C.H.A. Koster, *Affix Grammars*, in: J.E.L. Peck, editor, *Algol68 Implementation*, pp.95-109, North Holland Publishing, Amsterdam, 1971

[33] N.A. Lynch, R. Segala, F.W. Vaandrager and H.B. Weinberg, *Hybrid I/O automata*, in: R. Alur, T.A. Henzinger and E.D. Sontag, eds., *Hybrid Systems III*, Lecture Notes in Computer Science, Vol. 1066, pp.496-510, Springer-Verlag, 1996

[34] O. Maler, Z. Manna and A. Pnueli, *From timed to hybrid systems*, in: *Proceedings REX Workshop*, Lecture Notes in Computer Science, Vol. 600, pp.447-484, Springer-Verlag, 1992

[35] Mieke Massink, *Functional Techniques in Concurrency*, Ph.D. thesis, University of Nijmegen, February 1996

[36] H. Meijer, *Programmar: A Translator Generator*, Ph.D. thesis, University of Nijmegen, 1986

[37] C. van Reewijk, *tm: a code generator for structured data interfaces*, Espirit report E881/b38/TUD/CvR/8905, May 1989

[38] L.W.J. Rooijakkers, *Grammars in Funmath*, Techical Note, Faculty of Mathematics and Computing Science, University of Nijmegen, October 1991

[39] L.W.J. Rooijakkers, *Specifying infix operator association in Funmath*, Techical Note, Faculty of Mathematics and Computing Science, University of Nijmegen, July 1992

[40] L.W.J. Rooijakkers, *A Concrete Lexical Syntax for Funmath*, in *Documentatie bij Funmath Parser*, Technical Note CSI-N9401, Computing Science Institute, University of Nijmegen, December 1994

[41] Marc Seutter, *The development of semantic functions for a system description language with multiple interpretations*, Ph.D. thesis, University of Nijmegen, January 1994

[42] J.M. Spivey, *The Z notation — A Reference Manual*, Prentice-Hall, New York, 1989

[43] Huub van Thienen, *It's about time*, Ph.D. thesis, University of Nijmegen, January 1994

[44] D.A. Turner, *An Overview of Miranda*, ACM SIGPLAN Notices, December 1986

[45] William M. Waite and Gerhard Goos, *Compiler Construction*, Texts and Monographs in Computer Science, Springer-Verlag, 1984

[46] S. Warshall, *A theorem on Boolean matrices*, Journal of the ACM, Vol.9, pp.11-12, 1962

Samenvatting

De toepassing van digitale technologie in allerlei systemen neemt nog steeds toe. Steeds meer analoge electronische componenten worden vervangen door digitale componenten. Aanvankelijk werd deze verandering van technologie alleen gebruikt om bepaalde aspekten van bestaande systemen te verbeteren, zoals de produktiekosten, betrouwbaarheid, rekensnelheid en fysische aspekten zoals afmetingen en gewicht. De functionaliteit van deze eerste digitale systemen was echter gelijk aan die van hun analoge voorgangers. Tegenwoordig is de integratie van digitale componenten zover gevorderd dat veel moderne systemen zijn uitgerust met digitale programmeerbare processoren die bruikbaar zijn voor zeer diverse taken. Deze processoren worden vanwege hun diverse gebruiksmogelijkheden *general purpose* processoren genoemd. Verder bevatten de systemen software die nodig is om de processoren te programmeren voor hun specifieke taak in het systeem.

Deze ontwikkeling in de technologie heeft grote consequenties voor de theorie die nodig is bij het ontwerpen van systemen. De eerste toepassingen van digitale componenten waren aanleiding voor de ontwikkeling van discrete tijd modellen. In discrete tijdmodellen wordt de tijd opgedeeld tijdeenheden van gelijke lengte. Veranderingen in de toestand van een systeem kunnen alleen gebeuren op de grenzen van de tijdseenheden. Discrete tijdmodellen zijn een bijzonder geval van continue tijdmodellen, waarin het systeem op elk tijdstip van toestand kan veranderen. Analoge systemen worden beschreven met continue tijdmodellen. Zowel discrete als continue tijdmodellen zijn adequaat beschreven in klassieke wiskunde. Een bekend voorbeeld van het gebruik van wiskunde op dit gebied is de Fourier analyse en haar toepassing in digitale signaalbewerking.

Met de integratie van general purpose processoren in systemen is het ontwerp van de systeemsoftware een belangrijk deel geworden van het systeemontwerp. Er bestaan veel logica's om over de correctheid van programma's te redeneren. De digitale en analoge componenten van een systeem kunnen echter op een niet triviale manier met elkaar interageren. Om de correctheid van een dergelijk systeem aan te tonen moeten we daarom ook de *interface* tussen de heterogene componenten kunnen modelleren, en kunnen redeneren over de wisselwerking die plaatsvindt aan de interface. Op dit punt zijn de programmeertalen en logica's ontoereikend. De programmeertalen omdat ze geen constructies bevatten waarmee hardware kan worden beschreven, en de logica's omdat ze niet berekend zijn op het redeneren over continue systeemaspecten.

Funmath (*Fun*ctional *math*ematics is een specificatietaal die voortkomt uit de hardware specificatie wereld en deze tekortkomingen niet heeft. Het is mogelijk om in de taal

structuurbeschrijvingen van hardware te geven en deze structuren vervolgens in diverse wiskundige modellen te interpreteren. Op deze manier kunnen verschillende aspekten van een systeem onderzocht worden aan de hand van slechts één structuurbeschrijving. Het deel van Funmath dat de structuur van systemen beschrijft heet *Reals Real*izable *s*ystems). Verder is het ook mogelijk om in Funmath functionele programma's op te schrijven. Dit deel van de taal heet *Comma* (*Com*putational *ma*thematics).

Reals en Comma zijn *operationele* delen van Funmath. Een taal is operationeel als het alleen operationele beschrijvingen toelaat. Een beschrijving is operationeel als het direct kan worden afgebeeld op een realisatie van het beschreven systeem. Funmath laat ook beschrijvingen toe die niet operationeel zijn. Zulke beschrijvingen noemt men *declaratief*. Een taal die ook declaratieve beschrijvingen toelaat wordt declaratief genoemd. Declarativiteit is een belangrijke vereiste voor systeembeschrijvingstalen; het moet mogelijk zijn om systemen te beschrijven door het gewenste bedrag te specificeren. In het algemeen zijn dergelijke systeembeschrijvingen declaratief. Daarna kan zo'n declaratieve beschrijving stapsgewijs getransformeerd worden naar een operationele beschrijving, die afgebeeld kan worden op een systeemrealisatie die voldoet aan het gewenste gedrag. Dit proces kan volledig in Funmath worden uitgevoerd.

Dit proefschrift gaat in op twee aspekten van Funmath: de notatie en de typering. Beide aspekten zijn belangrijk bij het gebruik van Funmath voor theorie ontwikkeling. De notatie moet zo flexibel zijn dat theorieën op een gemakkelijk herkenbare manier kunnen worden opgeschreven. De typering moet een aantal vormen van polymorfisme ondersteunen die nodig zijn om algemene concepten zo te kunnen opschrijven dat ze in theorieën uit diverse toepassingsgebieden kunnen worden gebruikt.

Verder wordt de geschiktheid van Funmath als specificatietaal en als taal voor het ontwikkelen van theorie op het gebied van de informatica aangetoond.

In hoofdstuk 2 wordt de taal Funmath. Er wordt getoond hoe de gebruiker nieuwe notatie in de vorm van operatorpatronen kan introduceren. Verder wordt een algemeen mechanisme gepresenteerd waarmee de gebruiker de precedentie van infix operatoren kan regelen. Ook wordt een aantal basistypes en operatoren gegeven, waaronder operatoren voor het construeren van functietypes.

In hoofdstuk 3 worden verschillende vormen van polymorfe typering gedefiniëerd. Dit hoofdstuk is tevens een voorbeeld van het ontwikkelen van theorie in Funmath. Er wordt een model gegeven voor polymorfe functies en er worden operatoren gedefiniëerd voor het typeren van polymorfe functies. Deze type-operatoren zijn varianten van de operatoren uit hoofdstuk 2 die in combinatie met type-intersectie kunnen worden gebruikt voor het typeren van impliciet polymorfe functies en functies die ge-overload zijn.

In hoofdstuk 4 wordt een aantal veel gebruikte wiskundig concepten gedefiniëerd in Funmath, waaronder bekende begrippen uit domein theorie en algebra. Verder wordt een aantal begrippen uit de informatica gedefiniëerd, zoals lijsten, directe extensie en *pattern matching*. Deze begrippen worden echter op een meer algemene wiskundige wijze beschreven, zodanig dat ze ook toegepast kunnen worden in andere toepassingsgebieden.

In hoofdstuk 5 wordt een theorie van taalfuncties gepresenteerd, waarmee de structuur en betekenis van formele talen kan worden beschreven in Funmath. De basisoperatoren van

deze theorie zijn vereniging en concatenatie van talen. Met deze basisoperatoren kunnen contextvrije talen worden beschreven. Meer complexe talen worden beschreven met behulp van pattern matching en de directe extensie van vereniging van talen. Op deze manier kunnen contextsensitieve en zelfs ambigue talen worden beschreven. De theorie wordt vergeleken met attribuut en affix grammatica's en met semantische functies. Tenslotte wordt een aantal manieren getoond om ambiguïteiten in taalfuncties op te lossen.

Taalfuncties worden in hoofdstuk 6 gebruikt om een lexicale syntax en een formele grammatica voor Funmath te geven. De grammatica beschrijft ook de vorm van de operatoren die de gebruiker mag definiëren en het precedentie mechanisme voor infix operatoren.

In hoofdstuk 7 worden algoritmes gegeven voor het parseren van Funmath tekst. Voor het cruciale parseeralgortime, dat de operator precedentie relaties gebruikt om infix expressies te parseren, wordt een correctheidsbewijs gegeven.

Curriculum Vitae

van Frank van den Beuken

10 mei 1970
Geboren te Wanssum (Limburg).

17 juni 1988
Eindexamen Gymnasium B, Katholieke Scholengemeenschap "Jerusalem" voor HAVO en VWO te Venray.

31 augustus 1989
Propedeutisch examen Informatica (cum laude), Katholieke Universiteit Nijmegen.

22 mei 1992
Doctoraal examen Informatica (cum laude), Katholieke Universiteit Nijmegen.

1 juli 1992-31 september 1996
Assistent in opleiding bij het zwaartepunt Informatica voor Technische Toepassingen, Computing Science Institute, Katholieke Universiteit Nijmegen.

1 juni 1993
Behalen studieprijs voor Wiskunde en Informatica, Katholieke Universiteit Nijmegen.

1 oktober 1996-31 september 1997
Werkzaam als senior programmeur bij ICT Automatisering Eindhoven BV.